PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
NÚCLEO DE EDUCAÇÃO A DISTÂNCIA
Pós-graduação *Lato Sensu* em Ciência de Dados e Big Data

Leandro Marchezan do Nascimento Lopes

Modelo Preditivo de Preços de Apartamentos na cidade de São Paulo/SP

Belo Horizonte
2022

Leandro Marchezan do Nascimento Lopes

Modelo Preditivo de Preços de Apartamentos na cidade de São Paulo/SP

Trabalho de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados e Big Data como requisito parcial à obtenção do título de especialista.

Belo Horizonte
2022

Sumário

1. Introdução

1.1. Contextualização

Esse trabalho visa a elaboração de modelo de precificação de apartamentos na cidade de São Paulo-SP. A precificação é um processo para definir o valor monetário a ser cobrado por um produto, mercadoria ou serviço. Para definição do preço de determinado imóvel, são levados em consideração muitos fatores como: localização, quartos, vagas na garagem, idade do imóvel, bairro, área, estrutura do local entre outros. De acordo com a Associação Brasileira das Incorporadoras Imobiliárias (Abrainc), o tempo médio para a venda de um imóvel é de 1 ano e 4 meses. Assim, preços muito acima da média e não condizentes com o padrão do mercado afastam possíveis compradores, fazendo com que a venda demore muito mais. Um imóvel cujo preço de venda seja adequado se torna mais procurado, reduzindo o prazo para concretizar a venda. Em muitos casos, o proprietário tem dificuldade em definir o preço de venda, em geral por razões emocionais (valor sentimental pelo bem), por essa razão ter referências confiáveis é importante, sendo muito apropriado determinar um preço após pesquisar imóveis à venda com características semelhantes. Ademais, o tempo de venda pode variar, por conta de diversos fatores como localização, divulgação e estado de conservação sendo necessário pesquisar e saber os preços adequados. No outro lado, quem busca adquirir um apartamento corre o risco de pagar muito mais por um imóvel seja pela dificuldade na negociação ou pela falta de parâmetros em saber quando um imóvel está com um preço superestimado. Nessas circunstâncias, uma ferramenta de precificação auxiliaria no processo de avaliação imobiliária tendendo a reduzir o prazo de venda com a definição de um preço próximo ao que seria o preço de oferta ideal do imóvel. Os preços usados para os cálculos serão baseados em preços de anúncio e não nos preços pelos quais se efetivaram transações imobiliárias.

1.2. O problema proposto

O desenvolvimento do modelo de precificação visa auxiliar na determinação de um valor de venda adequado (preço ótimo) para o imóvel, de forma a não ter um sobrepreço que dificulte a negociação, reduzindo o tempo médio de venda e não ter um preço muito abaixo do mercado, orientando a parte vendedora quanto a aceitação de um preço mínimo do imóvel. Dessa forma, auxiliando quem pretende vender seu imóvel (pessoas físicas, construtoras e incorporadoras), e demais agentes envolvidos em transações imobiliárias, sejam os intermediários (imobiliárias e corretores) e compradores (consumidor final).

O principal objetivo deste trabalho é desenvolver um modelo preditivo dos preços de venda dos apartamentos anunciados e situados na Cidade de São Paulo. O conjunto de dados representa imóveis anunciados no mês de abril de 2019, sendo fornecidos o número de dormitórios, de banheiros, de vagas na garagem, área do imóvel e o distrito que se localiza o imóvel. Adicionando-se a essa base dados demográficos dos distritos da cidade (população, área em km², e densidade demográfica) e dados referentes ao nível de renda em relação ao salário mínimo em cada distrito. Ambos *datasets* obtidos no censo 2010. Para o tratamento do problema proposto, foram utilizados 3 *datasets,* sendo o primeiro obtido no repositório de dados da plataforma *kaggle*, e os demais dados no portal da Prefeitura de São Paulo.

1.3. Objetivos

Os objetivos dessa análise são:

- Realizar a análise descritiva preliminar dos dados a partir da junção das três bases de dados;

- Verificar a correlação entre os dados;

- Desenvolver modelos preditivos para os preços dos apartamentos colocados à venda na cidade de São Paulo – SP

O período a que se referem os dados de anúncios de venda dos apartamentos é de abril de 2019. Quanto aos modelos preditivos, 80 % dos dados foram utilizados como base de treinamento e 20% como dados de teste.

2. Coleta de Dados

Foram utilizados três *datasets* para a elaboração deste trabalho. O primeiro refere-se a um conjunto de dados obtidos na base de dados do *Kaggle* contendo cerca de 13.000 apartamentos para venda e aluguel na cidade de São Paulo, Brasil. Os dados provêm de várias fontes, especialmente sites de classificados de imóveis, obtido pelo seguinte link: https://www.kaggle.com/argonalyst/sao-paulo-real-estate-sale-rent-april-2019. Segue a descrição do formato desse dataset abaixo:

Nome da coluna/campo	Descrição	Tipo
Price	Preço final anunciado (R$ reais)	int64
Condo	Despesas do condomínio (valores desconhecidos são marcados como zero)	int64
Size	O tamanho da propriedade em Metros quadrados m² (somente áreas privadas)	int64
Rooms	Numero de quartos	int64
Toilets	Número de banheiros (todos os banheiros)	int64
Suites	Número de quartos com banheiro privativo (com suite)	int64
Parking	Número de vagas de estacionamento	int64
Elevator	Valor binário: 1 se houver elevador no prédio, 0 caso contrário	int64
Furnished	Valor binário: 1 se a propriedade for mobiliada, 0 caso contrário	int64
Swimming Pool	Valor binário: 1 se o imóvel tiver piscina, 0 caso contrário	int64
New	Valor binário: 1 se a propriedade for muito recente, 0 caso contrário	int64
District	O bairro e cidade onde o imóvel está localizado, ex:	object

	Itaim Bibi/São Paulo	
Negotiation Type	Venda ou aluguel.	object
Property Type	O tipo de propriedade	object
Latitude	Localização geográfica	float64
Longitude	Localização geográfica	float64

O segundo *dataset* refere-se a dados demográficos dos distritos pertencentes às Subprefeituras de São Paulo, contendo área (em km²), população no ano de 2010 e densidade demográfica (habitantes por quilômetro quadrado), obtidos no link: https://www.prefeitura.sp.gov.br/cidade/secretarias/subprefeituras/subprefeituras/dados_demograficos/index.php?p=12758.

Esses dados foram extraídos usando a biblioteca *Beautiful Soup* e posteriormente foi aplicada função específica para transformar os dados e retornar um *dataset* no formato descrito abaixo:

Nome da coluna/campo	**Descrição**	**Tipo**
Distrito	Distrito ou bairro da cidade	object
Área (km²)	Área do bairro em quilômetros quadrados	float64
População (2010)	Número de habitantes no bairro base censo 2010	float64
Densidade Demográfica (Hab/km²)	Relação entre as duas colunas anteriores	float64

Já o terceiro *dataset* refere-se a dados de domicílios por faixa de rendimento, medidos em salários mínimos, distribuídos entre as Subprefeituras e Distritos do município de São Paulo. Esses dados foram elaborados com base no censo de 2010. É um arquivo em formato *.xls* obtido diretamente no link abaixo, que está hospedado no site da prefeitura de São Paulo.

Segue a descrição do formato desse *dataset* abaixo:https://www.prefeitura.sp.gov.br/cidade/secretarias/licenciamento/desenvolvimento_urbano/dados_estatisticos/info_cidade/economia/index.php?p=260269. Segue link para *download*: https://www.prefeitura.sp.gov.br/cidade/secretarias/upload/Domicilios_faixa_rendimento_sal_minimos_2010.xls

Nome da coluna/campo	**Descrição**	**Tipo**
Distritos	Distrito ou Subprefeitura da cidade	object
Total	Total de domicílio particulares permanentes	object
Até 1/2	Classes de rendimento nominal mensal domiciliar Até 1/2 salário mínimo (SM).	object
Mais de 1/2 a 1	Classes de rendimento nominal mensal domiciliar Mais de 1/2 a 1 SM.	object
Mais de 1 a 2	Classes de rendimento nominal mensal domiciliar Mais de 1 a 2 SM.	object
Mais de 2 a 5	Classes de rendimento nominal mensal domiciliar Mais de 2 a 5 SM.	object
Mais de 5 a 10	Classes de rendimento nominal mensal domiciliar Mais de 5 a 10 SM.	object
Mais de 10 a 20	Classes de rendimento nominal mensal domiciliar Mais de 10 a 20 SM.	object
Mais de 20	Classes de rendimento nominal mensal domiciliar Mais de 20 SM.	object
Sem_rendimento	Classes de rendimento nominal mensal domiciliar sem rendimento.	object

2.1 Bibliotecas importadas

Foram importadas as seguintes bibliotecas para a execução dos códigos em Python no note-book:

-*pandas*: para fornecer estruturas de dados e ferramentas de análise;

-*numpy*: para o processamento numérico de matrizes e aplicação de funções matemáticas de alto nível;

-*matplotlib* e *seaborn*: para criação de gráficos bidimensionais e diversas configurações para

processamento de gráficos;

-*Scikit-learn*: biblioteca que fornecerá os algoritmos de aprendizado de máquina, métricas de avaliação dos modelos entre outras ferramentas para seleção e testagem de modelos;

-*Zipfile*: para trabalhar com arquivos ZIP;

-*BeautifulSoup*: biblioteca de extração de dados de arquivos HTML e XML

-*Folium*: Biblioteca que permite manipular os dados e visualizar o resultado imediatamente em um mapa interativo.

-*Wordcloud*: Biblioteca de representação gráfica(visualização) das palavras que aparecem com mais frequência em um texto de origem. Quanto maior a palavra no visual, mais comum é a palavra no documento;

-*Nominatim*: foi usada para encontrar locais na cidade de São Paulo por nome e endereço (geocodificação) retornando os dados de localização geográfica (latitude e longitude).

3. Processamento/Tratamento de Dados

Para iniciar o processamento dos dados, foram elaboradas algumas funções específicas para facilitar a execução de algumas atividades cuja descrição segue abaixo:

descompacta_zip(arq_descompactar) : Ao inserir um arquivo em formato .zip ela retorna o arquivo descompactado, descrevendo informações gerais dos arquivos descompactados.

extract_data_bairros(data): função criada para facilitar a manipulação dos dados extraídos com *BeautifulSoup* cuja aplicação é específica para compor o dataframe contendo informações sobre os distritos da cidade de São Paulo.

simetric(df, cols): função específica que converte os valores das colunas em escala logarítmica natural caso a assimetria seja menor com os registros nessa escala.

outlier_iqr(df,cols): função que retorna os valores limites máximos e mínimos para identificação de *outliers* usando o intervalo interquartílico (IQR).

plotar_grafico(y_t,y_pred): Função que cria um *scatter plot* com os resultados do modelo de *machine learning,* comparando os valores de teste e/ou de treino com os valores preditos. Retorna três gráficos, o primeiro é um gráfico dos valores reais e preditos, o segundo retorna a diferença dos valores reais e preditos (resíduos) e o terceiro gráfico plota a distribuição de frequência dos valores de resíduo.

3.1 Obtenção e estruturação dos Datasets

3.1.1 Primeiro Dataset – *df_imoveis*

Após salvar arquivo contendo os dados de imóveis em formato .zip na referida pasta de trabalho, aplicou-se a função 'descompacta_zip('sao-paulo-properties-april-2019.csv.zip')', que irá descompactar o arquivo do tipo *.zip*.

```
# Chamando função para descompactar .zip
descompacta_zip('sao-paulo-properties-april-2019.csv.zip')
```

```
File Name                                             Modified             Size
sao-paulo-properties-april-2019.csv          2019-10-21 14:34:46      1209935

'sao-paulo-properties-april-2019.csv.zip Descompactado'
```

A seguir, é realizada a abertura do arquivo carregando na variável "df".

```
# Abertura de arquivo e visualização dos primeiros registros:

csv_file = 'sao-paulo-properties-april-2019.csv'

df = pd.read_csv(csv_file, sep=',')

display(df.head())
df.info()
```

	Price	Condo	Size	Rooms	Toilets	Suites	Parking	Elevator	Furnished	Swimming Pool	New	District	Negotiation Type	Property Type	Latitude	Longitude
0	930	220	47	2	2	1	1	0	0	0	0	Artur Alvim/São Paulo	rent	apartment	-23.543138	-46.479486
1	1000	148	45	2	2	1	1	0	0	0	0	Artur Alvim/São Paulo	rent	apartment	-23.550239	-46.480718
2	1000	100	48	2	2	1	1	0	0	0	0	Artur Alvim/São Paulo	rent	apartment	-23.542818	-46.485665
3	1000	200	48	2	2	1	1	0	0	0	0	Artur Alvim/São Paulo	rent	apartment	-23.547171	-46.483014
4	1300	410	55	2	2	1	1	1	0	0	0	Artur Alvim/São Paulo	rent	apartment	-23.525025	-46.482436

Verifica-se que há dados do tipo *int64, float64 e object* totalizando 16 colunas e 13640 registros não nulos. Também que há registros de imóveis para aluguel e que na coluna do distrito é incluída a cidade a qual pertence. A seguir é feita a verificação de quais os tipos de negociação constantes no dataframe:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13640 entries, 0 to 13639
Data columns (total 16 columns):
Price                13640 non-null int64
Condo                13640 non-null int64
Size                 13640 non-null int64
Rooms                13640 non-null int64
Toilets              13640 non-null int64
Suites               13640 non-null int64
Parking              13640 non-null int64
Elevator             13640 non-null int64
Furnished            13640 non-null int64
Swimming Pool        13640 non-null int64
New                  13640 non-null int64
District             13640 non-null object
Negotiation Type     13640 non-null object
Property Type        13640 non-null object
Latitude             13640 non-null float64
Longitude            13640 non-null float64
dtypes: float64(2), int64(11), object(3)
memory usage: 1.7+ MB
```

```
# Checando valores únicos para tipo de negociação (será extraído apenas os registros de venda)

print('Tipo de negociação:', list(df['Negotiation Type'].unique()))
```

```
Tipo de negociação: ['rent', 'sale']
```

Após são selecionados apenas os imóveis destinados para venda, excluídos os registros duplicados e renomeadas as colunas para português. Também é corrigida a informação da coluna 'Distrito', extraindo o nome da cidade. É feito comando para verificar a quais cidades e tipos de imóveis se referem os anúncios. Por fim, ao ter registros excluídos, é feito o 'reset' do index e exibida nova *info*.

Pela imagem abaixo, verifica-se que o dataframe possui 6302 registros de venda de apartamentos na cidade de São Paulo-SP. Dessa forma, por não serem mais úteis, serão excluídas as colunas “Cidade” e “Tipo_imovel”, totalizando 14 colunas no dataframe.

```
# Seleção dos imóveis para venda:
df_imoveis = df.loc[df['Negotiation Type']== 'sale']
df_imoveis.drop(columns=['Negotiation Type'], inplace=True)

# Exclusão de dados duplicados, renomeação das colunas e atualização do index
df_imoveis = df_imoveis.drop_duplicates()
df_imoveis.rename(columns={'Price': 'Preco', 'Size':'Area', 'Rooms': 'Quartos', 'Toilets':'Banheiros',
                           'Parking': 'Garagem', 'Elevator': 'Elevador', 'Furnished': 'Mobiliado', 'Swimming Pool': 'Piscina',
                           'New': 'Novo', 'District': 'Distrito', 'Property Type':'Tipo_imovel'}, inplace=True)

# Como verificado, a coluna "Distrito" contém nome do distrito e cidade.
# Fatiamento dessa coluna em 'Distrito' e 'Cidade':
df_imoveis[['Distrito','Cidade']] = df_imoveis['Distrito'].str.split('/', expand = True)
df_imoveis.reset_index(drop=True, inplace=True)

# Verificação se todos os registros são da cidade de São Paulo e quais os tipos de imóveis que compões o DataFrame:
print(' Cidades no dataframe:', list(df_imoveis['Cidade'].unique()), '\n',
      'Tipos de imóveis: ', list(df_imoveis['Tipo_imovel'].unique()), '\n')

df_imoveis.info()
```

```
 Cidades no dataframe: ['São Paulo']
 Tipos de imóveis:  ['apartment']

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6302 entries, 0 to 6301
Data columns (total 16 columns):
Preco          6302 non-null int64
Condo          6302 non-null int64
Area           6302 non-null int64
Quartos        6302 non-null int64
Banheiros      6302 non-null int64
Suites         6302 non-null int64
Garagem        6302 non-null int64
Elevador       6302 non-null int64
Mobiliado      6302 non-null int64
Piscina        6302 non-null int64
Novo           6302 non-null int64
Distrito       6302 non-null object
Tipo_imovel    6302 non-null object
Latitude       6302 non-null float64
Longitude      6302 non-null float64
Cidade         6302 non-null object
dtypes: float64(2), int64(11), object(3)
memory usage: 787.8+ KB
```

Exclusão das colunas ‘Cidade’ e ‘Tipo_imovel’:

```
# Como os registros são únicos, essas colunas serão excluídas

df_imoveis.drop(columns=['Cidade','Tipo_imovel'], inplace=True)
df_imoveis.head()
```

	Preco	Condo	Area	Quartos	Banheiros	Suites	Garagem	Elevador	Mobiliado	Piscina	Novo	Distrito	Latitude	Longitude
0	732600	1000	74	1	2	1	2	1	0	1	0	Vila Madalena	-23.552129	-46.692244
1	1990000	2400	164	4	5	2	3	1	1	1	0	Vila Madalena	-23.551613	-46.699106
2	720000	700	70	2	2	1	1	1	0	1	1	Vila Madalena	-23.547687	-46.692594
3	1680000	1580	155	3	5	3	2	1	0	1	0	Vila Madalena	-23.552590	-46.691104
4	1200000	900	56	2	2	1	2	0	1	1	0	Vila Madalena	-23.553087	-46.697890

Visualizando os dados do dataframe *df_imoveis*, checando registros nulos e descrição sumária, percebe-se que, apesar de não haver dados nulos, há registros de valor zero para valor de Condomínio, sendo necessário aprofundar a análise nesses registros. As variáveis Preço, Condo e Area são variáveis quantitativas contínuas, as variáveis Quartos, Banheiros, Suítes e Garagem correspondem a variáveis quantitativas discretas e, por fim, as variáveis Elevador, Mobiliado, Piscina e Novo correspondem a variáveis qualitativas binárias.

```
# Descrição sumária dos registros do Dataframe:
display(df_imoveis.describe())
# Checagem de valores Nan:
print('\n',"Total de valores NaN: ", df_imoveis.isna().sum().sum())
```

	Preco	Condo	Area	Quartos	Banheiros	Suites	Garagem	Elevador	Mobiliado	Piscina
unt	6.302000e+03	6302.000000	6302.000000	6302.000000	6302.000000	6302.000000	6302.000000	6302.000000	6302.000000	6302.000000
ean	6.131756e+05	542.372739	79.002856	2.324976	2.043002	0.935100	1.331482	0.415582	0.118692	0.543478
std	7.448893e+05	627.052401	51.081138	0.713803	0.920705	0.773967	0.755085	0.492861	0.323452	0.498146
min	4.200000e+04	0.000000	30.000000	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	2.500000e+05	161.250000	50.000000	2.000000	2.000000	1.000000	1.000000	0.000000	0.000000	0.000000
50%	3.808285e+05	400.000000	62.000000	2.000000	2.000000	1.000000	1.000000	0.000000	0.000000	1.000000
75%	6.800000e+05	700.000000	88.000000	3.000000	2.000000	1.000000	2.000000	1.000000	0.000000	1.000000
max	1.000000e+07	8920.000000	620.000000	6.000000	7.000000	6.000000	7.000000	1.000000	1.000000	1.000000

```
Total de valores NaN:  0
```

3.1.2 Segundo Dataset – df_bairros

O segundo dataset foi estruturado a partir do uso da biblioteca Beautiful Soup do site https://www.prefeitura.sp.gov.br/cidade/secretarias/subprefeituras/subprefeituras/dados_demograficos/index.php?p=12758. Extraindo-se as informações de texto de interesse como mostrado na figura abaixo, posteriormente foi aplicada função específica – extract_data_bairros(data) - a qual foi estruturada para retornar um Dataframe a partir do arquivo extraído com *Beautiful Soup*.

```
# Função para buscar dados dos bairros e subprefeituras da cidade de São Paulo:
# <'https://www.prefeitura.sp.gov.br/cidade/secretarias/subprefeituras/subprefeituras/dados_demograficos/index.php?p=12758'>

# Uso de Beautiful Soup

url = 'https://www.prefeitura.sp.gov.br/cidade/secretarias/subprefeituras/subprefeituras/dados_demograficos/index.php?p=12758'
html_text = requests.get(url).text
soup = BeautifulSoup(html_text, 'html.parser')
title = soup.find('tbody').get_text()
data=title.strip().split('\n\n')
data
```

```
['Subprefeituras\nDistritos\nÁrea (km²)\nPopulação (2010)\nDensidade Demográfica (Hab/km²)',
 '\nAricanduva\nAricanduva\n6,60\n89.622\n13.579',
 '\nCarrão\n7,50\n83.281\n11.104',
 '\nVila Formosa\n7,40\n94.799\n12.811',
 '\nTOTAL\n21,50\n267.702\n12.451',
 '\nButantã\nButantã\n12,50\n54.196\n4.336',
 '\nMorumbi\n11,40\n46.957\n4.119',
 '\nRaposo Tavares\n12,60\n100.164\n7.950',
 '\nRio Pequeno\n9,70\n118.459\n12.212',
 '\nVila Sônia\n9,90\n108.441\n10.954',
 '\nTOTAL\n56,10\n428.217\n7.633',
 '\nCampo Limpo\nCampo Limpo\n12,80\n211.361\n16.513',
```

```
# Criação do Dataframe df_bairros

df_bairros = extract_data_bairros(data)

df_bairros.to_csv("df_bairros.csv")

type(df_bairros)
```

```
Extração de dados está CORRETA
Cada lista possui 97 elementos

pandas.core.frame.DataFrame
```

Visualizando dos dados usando *.head()* e *.tail()*:

```
# Criação do Dataframe df_bairros
display(df_bairros.head())
display(df_bairros.tail())
```

	Distrito	Área (km²)	População (2010)	Densidade Demográfica (Hab/km²)
1	Aricanduva	6,60	89.622	13.579
2	Carrão	7,50	83.281	11.104
3	Vila Formosa	7,40	94.799	12.811
4	Butantã	12,50	54.196	4.336
5	Morumbi	11,40	46.957	4.119

	Distrito	Área (km²)	População (2010)	Densidade Demográfica (Hab/km²)
92	Moema	9,00	83.368	9,263
93	Saúde	8,90	130.780	14,694
94	Vila Mariana	8,60	130.484	15,173
95	São Lucas	9,90	142.347	14,378
96	Vila Prudente	9,90	104.242	10,529

Pelos registros do dataframe *df_bairros,* verifica-se que é necessário tratar o separador decimal. Por padrão será usado o ponto(.) no lugar da vírgula(,) e vazio() no lugar do ponto(.). Abaixo segue imagem dos comandos necessários para esses ajustes.

```
# Uso de função lambda e .replace para mudar vírgulas por pontos e pontos e vírgulas por espaço vazio

df_bairros['Área (km²)'] = df_bairros['Área (km²)'].apply(lambda x: float(x.replace(",",".")))
df_bairros['População (2010)'] = df_bairros['População (2010)'].apply(lambda x: float(x.replace(".","")))
df_bairros['Densidade Demográfica (Hab/km²)'] = df_bairros['Densidade Demográfica (Hab/km²)'].apply(lambda x: float(x

# Visualizando as 'pontas' do Dataframe
display(df_bairros.head())
display(df_bairros.tail())
```

	Distrito	Área (km²)	População (2010)	Densidade Demográfica (Hab/km²)
1	Aricanduva	6.6	89622.0	13579.0

Visualizando sumariamente os dados do df_bairros, não há registros nulos, portanto, não há necessidade de maiores tratamentos no momento.

```
# Descrição sumária do Dataframe:
display(df_bairros.info())
display(df_bairros.describe())
print('\n',"Valores NaN:",'\n\n', df_bairros.isna().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 96 entries, 1 to 96
Data columns (total 4 columns):
Distrito                            96 non-null object
Área (km²)                          96 non-null float64
População (2010)                    96 non-null float64
Densidade Demográfica (Hab/km²)     96 non-null float64
dtypes: float64(3), object(1)
memory usage: 3.8+ KB

None
```

	Área (km²)	População (2010)	Densidade Demográfica (Hab/km²)
count	96.000000	96.000000	96.000000
mean	15.718750	117223.989583	11035.541667
std	26.395571	69639.133760	5221.350017
min	2.100000	8258.000000	41.000000
25%	7.350000	69368.000000	7709.750000
50%	9.900000	104594.500000	10923.000000
75%	13.350000	142353.250000	14169.500000
max	200.000000	360787.000000	26715.000000

```
 Valores NaN:

 Distrito                            0
Área (km²)                          0
População (2010)                    0
Densidade Demográfica (Hab/km²)     0
dtype: int64
```

3.1.3 – Terceiro Dataset – df_rendas:

Abrindo o arquivo obtido no site da prefeitura (Domicilios_faixa_de_rendimento) e salvando na pasta de trabalho do jupyter notebook. É necessário ajustar a partir de que linha

deve começar e terminar o dataframe (uso do método *.read_excel*() usando os parâmetros *skiprows* e *skipfooter*). Também é necessário renomear algumas colunas.

Dataframe - Renda por Domicílio

```
# Abrindo arquivo .xlsx e renomeando colunas

df_rendas = pd.read_excel('Domicilios_faixa_rendimento_sal_minimos_2010.xls', skiprows=6, skipfooter=5)

df_rendas.rename(columns = {'Unnamed: 0' : 'Distrito', 'Unnamed: 1': 'Total_domicilios',
                            "Sem rendimento (3)": "Sem_rendimento"}, inplace=True)

# Renomeando a última coluna:
cols = df_rendas.columns
for col in cols:
    if col.startswith('Sem'):
        df_rendas.rename(columns = {col:"Sem_rendimento"}, inplace=True)
df_rendas.columns
```

```
Index(['Distrito', 'Total_domicilios', 'Até 1/2', 'Mais de 1/2 a 1',
       'Mais de 1 a 2', 'Mais de 2 a 5', 'Mais de 5 a 10', 'Mais de 10 a 20',
       'Mais de 20', 'Sem_rendimento'],
      dtype='object')
```

Verifica-se que as colunas são do tipo *object*, apesar de conterem valores numéricos. Visualizando os primeiros e últimos registros, percebe-se a necessidade de substituir vírgulas por pontos e pontos por vazio.

```
# Visualizando as 'pontas' do Dataframe
display(df_rendas.head())
display(df_rendas.tail())
```

	Distrito	Total_domicilios	Até 1/2	Mais de 1/2 a 1	Mais de 1 a 2	Mais de 2 a 5	Mais de 5 a 10	I
0	São Paulo	3.574.286	20.129	225.166	588.778	1.212.485	714.900	
1	Aricanduva/Formosa/Carrão	85.188	197	4.788	11.237	28.095	21.081	
2	Aricanduva	27.661	90	1.996	4.457	10.327	6.550	
3	Carrão	27.115	42	1.266	2.908	8.239	7.254	
4	Vila Formosa	30.412	65	1.526	3.872	9.529	7.277	

	Distrito	Total_domicilios	Até 1/2	Mais de 1/2 a 1	Mais de 1 a 2	Mais de 2 a 5	Mais de 5 a 10
123	Vila Mariana	51.822	77	549	1.678	7.032	11.350
124	Vila Prudente/Sapopemba	165.163	1.225	11.958	29.924	63.634	34.896
125	São Lucas	45.770	135	2.593	6.607	16.480	12.162
126	Sapopemba	84.686	721	7.606	18.943	35.860	14.426
127	Vila Prudente	34.707	369	1.759	4.374	11.294	8.308

Devido a frequência duplicada de alguns registros, foi verificado que alguns registros ou se referiam ao distrito ou a subprefeitura (a qual compreende vários distritos). Assim, foi realizada a exclusão manual dos registros que não interessam para análise e modificado os registros escritos incorretamente. Também foi identificada a necessidade de tratar o separador decimal da mesma forma que no “df_bairros”:

```
#Correção Registro Distrito 'São Miguel' e 'Moóca'

df_rendas.iloc[[105],[0]]='São Miguel'

df_rendas.loc[df_rendas['Distrito'] == 'Moóca', 'Distrito']='Mooca'
```

```
# Exclusão de registros manualmente.
#Ex.: 2 Registros Butantã: um é o distrito e o outro se refere a Subprefeitura
index_drop= [0,1,5,11,15,19,23,28,31,34,37,41,44,49,51,54,61,64,71,74,79,82,87,91,95,99,103,107,116,120,124]

df_rendas.drop(index=index_drop, inplace=True)
df_rendas.drop_duplicates(inplace=True)
# Correção de pontos e vírgulas para números e conversão para percentagem
cols = df_rendas.columns

for col in cols:
    if (col != 'Distrito'):
        df_rendas[col] = df_rendas[col].apply(lambda x: float(x.replace(".","")))

print("Total de registros df_rendas: ",len(list(df_rendas['Distrito'])))
```

```
Total de registros df_rendas:  96
```

Para fins de análise de dados, os registros referentes a quantidade de domicílios serão convertidos para valores percentuais em relação a coluna *“Total_domicílios”*, preservando as colunas “*Distrito*” e “*Total_domicílios*”, conforme mostra a imagem abaixo:

```
# Conversão para percentagem

for col in cols:
    if (col != 'Distrito') and (col != 'Total_domicilios'):
        df_rendas[col+'_SM_%'] = round(df_rendas[col]/df_rendas['Total_domicilios']*100,2)
        df_rendas.drop(columns=col, inplace=True)
```

Segue descrição sumária e contagem de valores nulos do dataframe *“*df_rendas”. Foi verificado sendo que não há registros nulos ou faltantes, logo, não há necessidade de tratamento para esse tipo de problema nesses dados.

```
# Descrição sumária do Dataframe e checagem de valores Nan:
display(df_rendas.info())
display(df_rendas.describe())
print("Total de valores NaN: ", df_rendas.isna().sum().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 103 entries, 1 to 127
Data columns (total 10 columns):
Distrito                103 non-null object
Total_domicilios        103 non-null float64
Até 1/2_SM_%            103 non-null float64
Mais de 1/2 a 1_SM_%    103 non-null float64
Mais de 1 a 2_SM_%      103 non-null float64
Mais de 2 a 5_SM_%      103 non-null float64
Mais de 5 a 10_SM_%     103 non-null float64
Mais de 10 a 20_SM_%    103 non-null float64
Mais de 20_SM_%         103 non-null float64
Sem_rendimento_SM_%     103 non-null float64
dtypes: float64(9), object(1)
memory usage: 8.9+ KB

None
```

	Total_domicilios	Até 1/2_SM_%	Mais de 1/2 a 1_SM_%	Mais de 1 a 2_SM_%	Mais de 2 a 5_SM_%	Mais de 5 a 10_SM_%	Mais de 10 a 20_SM_%
count	103.000000	103.000000	103.000000	103.000000	103.000000	103.000000	103.000000
mean	42020.388350	0.528058	5.779903	14.948641	32.395243	21.139709	12.095631
std	27204.655742	0.425985	3.022549	7.064503	9.225873	5.086254	7.838326
min	2349.000000	0.090000	0.720000	1.960000	8.720000	7.880000	0.930000
25%	26536.500000	0.235000	3.500000	9.180000	27.090000	18.140000	5.680000
50%	35554.000000	0.400000	5.630000	15.140000	35.770000	21.900000	11.110000
75%	47985.000000	0.690000	7.410000	19.710000	39.285000	25.295000	17.690000
max	165163.000000	3.150000	17.670000	27.800000	45.710000	29.220000	26.950000

```
Total de valores NaN:  0
```

3.2. Junção de datasets

3.2.1 Primeira junção de Datasets: df_bairros e df_rendas:

Nessa seção é apresentada a primeira junção de dataframes usando como chave a coluna “Distrito”. Será aplicado o método *.merge()* usando *inner join* como parâmetro entre os *dataframes* “df_bairros” e “df_rendas” criando um novo dataframe (“df_distritos”). O total de registros do novo dataframe deve ser igual ao “df_bairros”. Também já há a verificação de valores nulos.

df_bairros e df_rendas

```
# Merge dos Dataframes df_bairros e df_rendas, inner join e contagem do número de registros
df_distritos = pd.merge(df_bairros, df_rendas, on='Distrito', how='inner')
print(' Total de registros em df_distritos: ', len(df_distritos), '\n',
      'Total de registros em df_bairros:   ',len(df_bairros),'\n',
      'Total de registros em df_rendas:    ',len(df_rendas))

# Checar Nan's nas colunas e percentual de Nan's em cada coluna
print(" Total de Valores NaNs:              ", df_distritos.isna().sum().sum())
```

```
 Total de registros em df_distritos:  96
 Total de registros em df_bairros:    96
 Total de registros em df_rendas:    96
 Total de Valores NaNs:               0
```

3.2.1 Segunda junção de Datasets: df_imoveis e df_distritos:

Realização da segunda junção de dataframes usando como chave o campo “Distrito”. Será aplicado novamente o método *.merge()* usando *left join* como parâmetro entre os datasets “df_imoveis” e “df_distritos” criando um novo dataframe (df). O total de registros do novo dataframe será necessariamente igual ao “df_imoveis”. Foi aplicada regra para verificar a identificação dos distritos e, caso contrário, retornar quantos e quais distritos foram excluídos da junção, uma vez que ao aplicar *left join,* caso não encontre correspondência, irá devolver valor nulo nas colunas correspondentes ao dataframe “df_distritos”.

df_imoveis e df_distritos

```
# Merge Dataframes df_imoveis e df_distritos e posterior verificação pois a chave "Distrito" em df_imoveis pode conter erros

df = pd.merge(df_imoveis, df_distritos, on = 'Distrito', how = 'left')
l_distr = df['Distrito'].loc[df['População (2010)'].isna()].unique()
print(' Total de Distritos não identificados:', len(l_distr),'\n','Distritos não identificados: ', l_distr)
```

```
 Total de Distritos não identificados: 6
 Distritos não identificados:  ['Vila Madalena' 'Brooklin' 'Vila Olimpia' 'Medeiros' 'Guaianazes'
 'Jardim São Luis']
```

Após verificar quais distritos não foram identificados, constatou-se a necessidade de correção gramatical em alguns distritos e a atualização de alguns lugares no “df_imoveis”

pois foram registrados pelo nome da vila ou bairro e não pelo nome oficial do distrito (por exemplo, vila madalena pertence ao distrito de Pinheiros).

```
# Correção gramatical dos nomes dos distritos

df_distritos.loc[df_distritos['Distrito'] == 'Guaianases', 'Distrito']='Guaianazes'

df_imoveis.loc[df_imoveis['Distrito'] == 'Jardim São Luis', 'Distrito']='Jardim São Luís'
df_imoveis.loc[df_imoveis['Distrito'] == 'Medeiros', 'Distrito']='Vila Medeiros'

# correção de lugares:df_imoveis com local identificado pelo nome da vila ou bairro e df_distritos pelo nome oficia
df_imoveis.loc[df_imoveis['Distrito'] == 'Vila Madalena', 'Distrito']='Pinheiros'
df_imoveis.loc[df_imoveis['Distrito'] == 'Brooklin', 'Distrito']='Itaim Bibi'
df_imoveis.loc[df_imoveis['Distrito'] == 'Vila Olimpia', 'Distrito']='Itaim Bibi'
```

Após esses ajustes, é definitivamente aplicado o *.merge()* criando o dataframe “df”, verificando novamente se há algum distrito não identificado, eliminando eventuais valores nulos e resetando o index usando *.set_index()*, conforme imagem abaixo:

```
# Merge com adequação dos nomes dos Distritos

df = pd.merge(df_imoveis, df_distritos, on = 'Distrito', how = 'left')

l_distr = df['Distrito'].loc[df['População (2010)'].isna()].unique()
print(' Total de Distritos não identificados:', len(l_distr),'\n','Distritos não identificados: ', l_distr)

df.dropna(inplace=True)
df.reset_index(drop = True,inplace=True)

print( " Total de valores nulos:", df.isna().sum().sum(),'\n Dimensão df:', df.shape)
```

```
 Total de Distritos não identificados: 0
 Distritos não identificados:  []
 Total de valores nulos: 0
 Dimensão df: (6302, 26)
```

4. Análise Exploratória dos dados

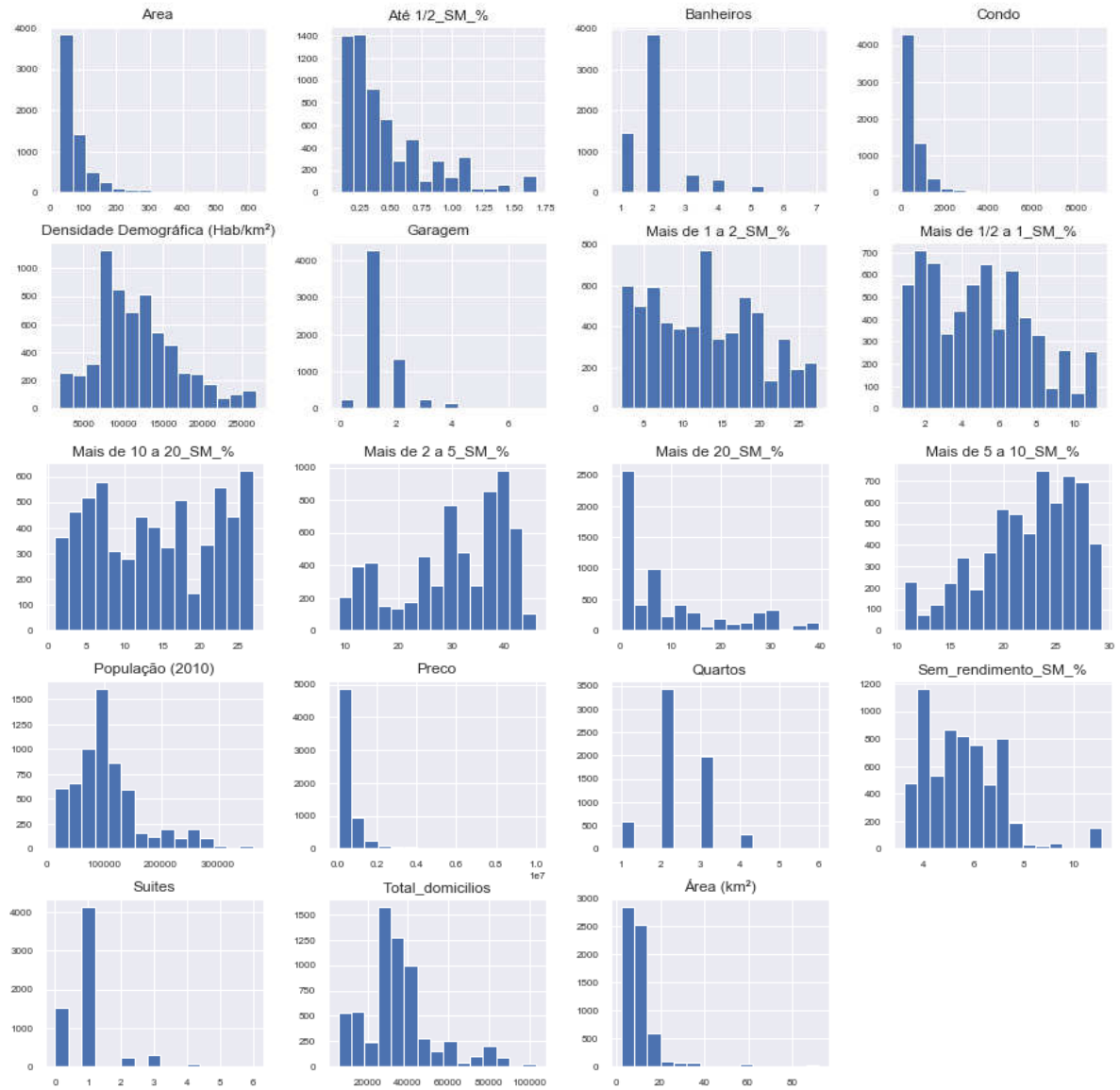
Agora o dataset principal está pronto(dataframe “df”), a partir dele será feita análise exploratória de dados e , caso haja necessidade, análise minuciosa e mais profunda de determinadas variáveis. Aplicando método .info() no “df” , verifica-se que há 6302 registros não nulos de imóveis a venda com informações distribuídas em 26 colunas, sendo 14 do tipo *float64*, 11 do tipo *int64* e 1 do tipo *object*.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6302 entries, 0 to 6301
Data columns (total 26 columns):
Preco                                6302 non-null int64
Condo                                6302 non-null int64
Area                                 6302 non-null int64
Quartos                              6302 non-null int64
Banheiros                            6302 non-null int64
Suites                               6302 non-null int64
Garagem                              6302 non-null int64
Elevador                             6302 non-null int64
Mobiliado                            6302 non-null int64
Piscina                              6302 non-null int64
Novo                                 6302 non-null int64
Distrito                             6302 non-null object
Latitude                             6302 non-null float64
Longitude                            6302 non-null float64
Área (km²)                           6302 non-null float64
População (2010)                     6302 non-null float64
Densidade Demográfica (Hab/km²)      6302 non-null float64
Total_domicilios                     6302 non-null float64
Até 1/2_SM_%                         6302 non-null float64
Mais de 1/2 a 1_SM_%                 6302 non-null float64
Mais de 1 a 2_SM_%                   6302 non-null float64
Mais de 2 a 5_SM_%                   6302 non-null float64
Mais de 5 a 10_SM_%                  6302 non-null float64
Mais de 10 a 20_SM_%                 6302 non-null float64
Mais de 20_SM_%                      6302 non-null float64
Sem_rendimento_SM_%                  6302 non-null float64
dtypes: float64(14), int64(11), object(1)
memory usage: 1.3+ MB
```

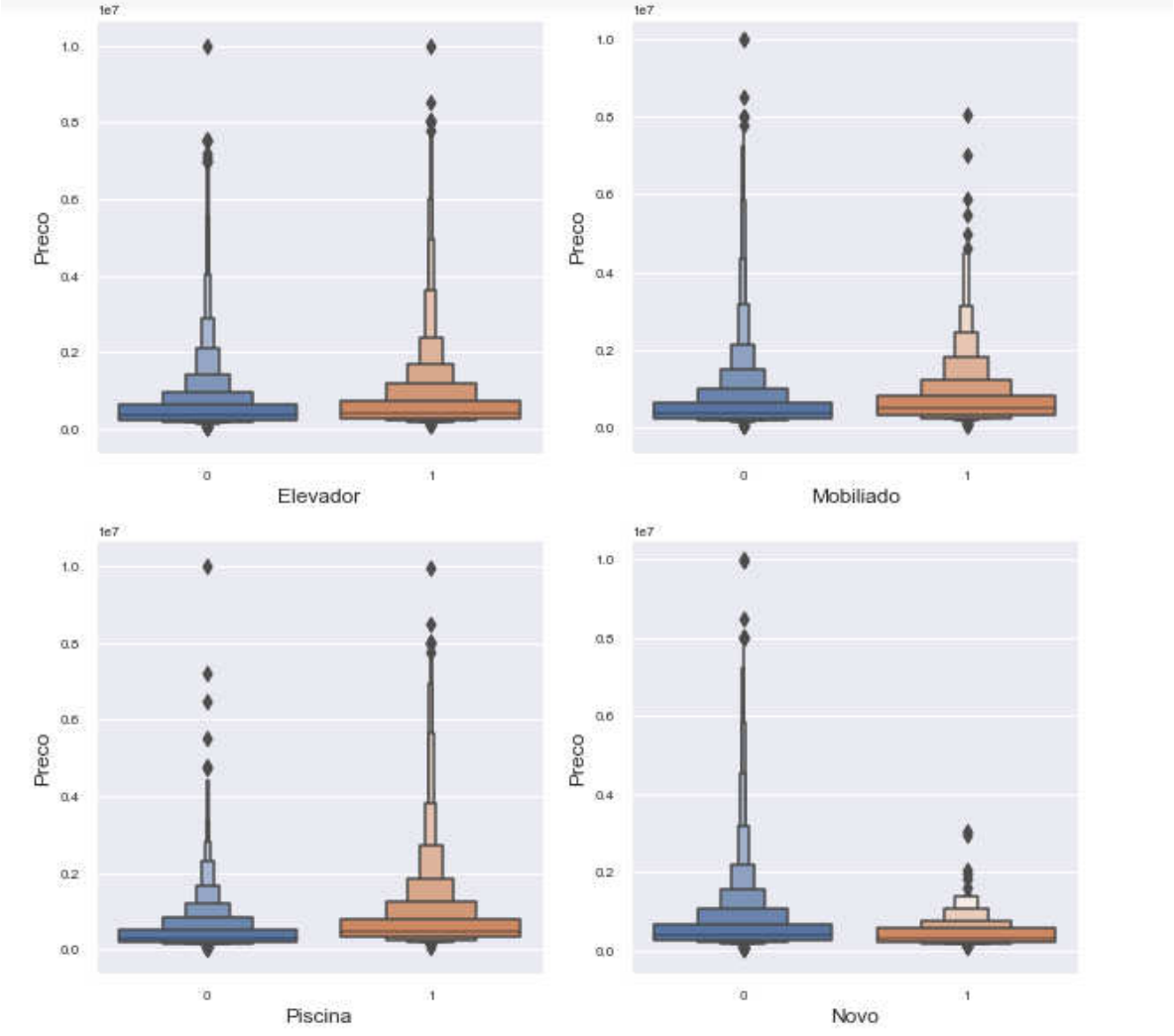
Visualizando a distribuição dos dados das variáveis quantitativas do Dataframe através de histograma, percebe-se que há variáveis com nítida distribuição assimétrica à esquerda como “Area”, “Banheiros”, “Condo” por exemplo:

```
# Visualização da distribuição dos dados do Dataframe

l_columns = list(df.columns[0:7])+ list(df.columns[14:])
hist = df[l_columns].hist(bins=15, figsize=(15,15))
```

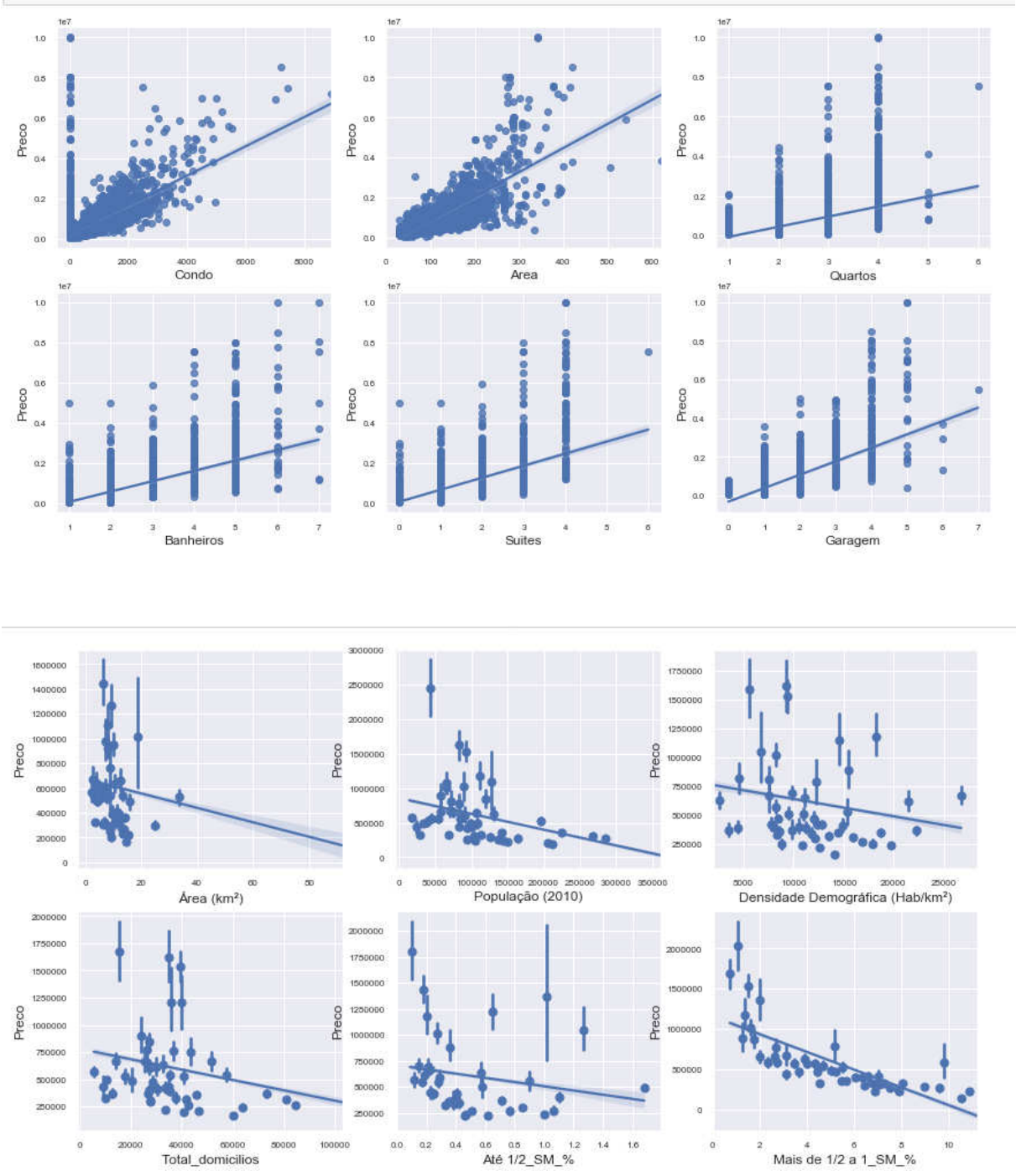


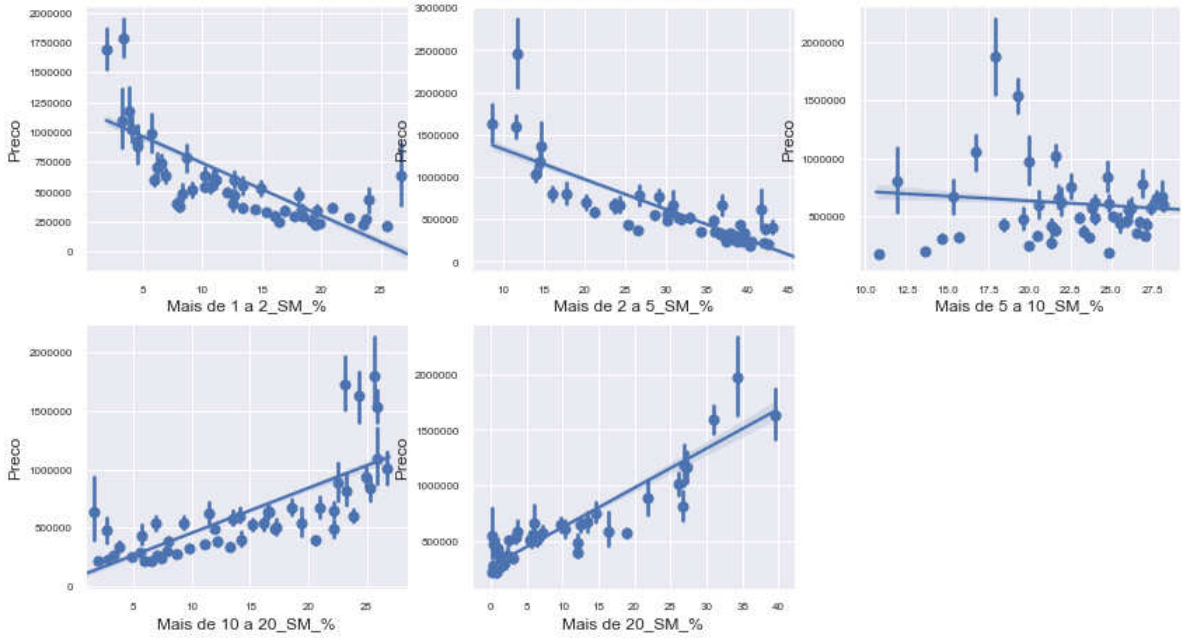
A seguir serão visualizadas as distribuições das variáveis binárias em relação ao preço do imóvel.

```
# Continuando a exploração dos dados: Relação do Preço com as variáveis: elevador, mobiliado, piscina e novo
l_columns = df.columns[7:11]
pos = 0
plt.figure(figsize = (10,10))
for i in l_columns:
    pos +=1
    plt.subplot(2,2,pos)
    ax = sns.boxenplot(x = i , y = 'Preco',data = df )
```



Percebe-se que as variáveis binárias Elevador e Mobiliado não variam conforme o preço do imóvel, já o imóvel ser novo ou possuir piscina implicam numa tendência de não ultrapassar uma certa faixa de preço, especialmente se for um imóvel novo. A seguir foi plotado gráfico de correlação entre as variáveis preditoras e a variável resposta (Preço):





Verifica-se que a maioria das variáveis preditoras possuem algum grau de correlação com a variável Preço. Também se nota uma expressividade de valores zero na variável condomínio, considerando que o dataframe possui somente registro de apartamentos, e que todo prédio possui taxa condominial, logo, esses registros são valores desconhecidos ou faltantes (*missing values*).

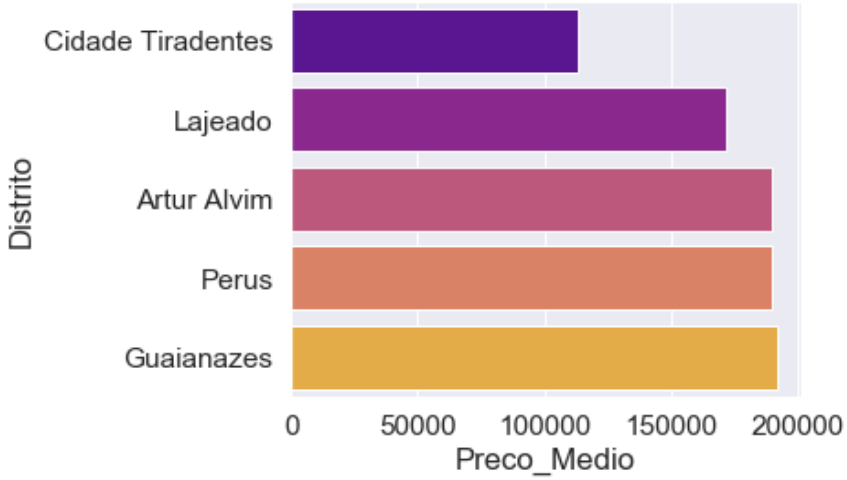
Em relação as variáveis geográficas (Latitude e Longitude), verifica-se a existência de muitos valores inconsistentes, que não correspondem a área da cidade de São Paulo. Abaixo segue gráfico de dispersão entre essas variáveis, o esperado é que formem uma imagem aproximada da área urbana da cidade, mas ficam marcados pontos esparsos.

A seguir foi usada a biblioteca “folium” para visualizar a distribuição geográfica dos imóveis. É possível verificar que uma quantidade razoável de registros não está localizada na cidade de São Paulo. Do total de 6302 registros, pode-se considerar que 5835 contém dados de latitude e longitude adequadamente registrados.

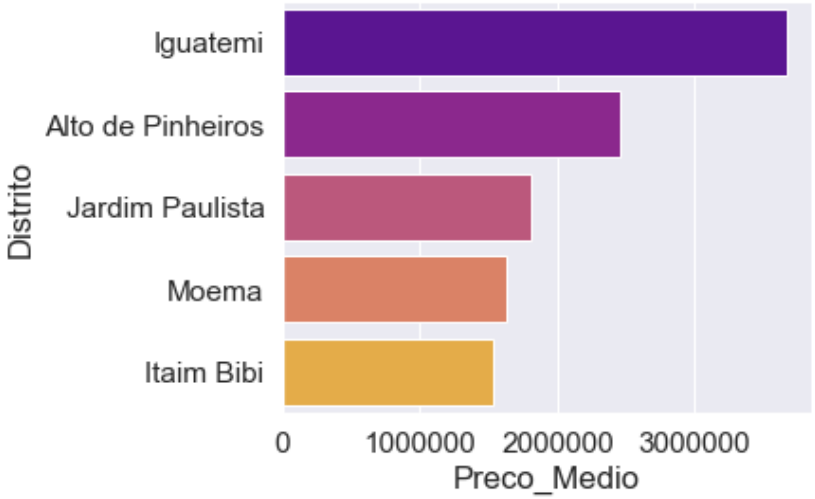
```
# Uso biblioteca folium usando colunas Latitudes e Longitudes informadas:

m = folium.Map(
    location=[-23.3, -46.6],
    tiles='Stamen Toner',
    zoom_start=8
)
mc = MarkerCluster()

for index, value in df.iterrows():
  mc.add_child(folium.Marker([value['Latitude'], value['Longitude']],
                popup=str(value['Distrito']),
                tooltip=value['Distrito'],
                icon=folium.Icon(icon='book'))).add_to(m)
m
```

Abaixo segue relação dos distritos com os cinco menores e cinco maiores preços médios de apartamentos, respectivamente:

Distritos com os menores preço médio de apartamentos:

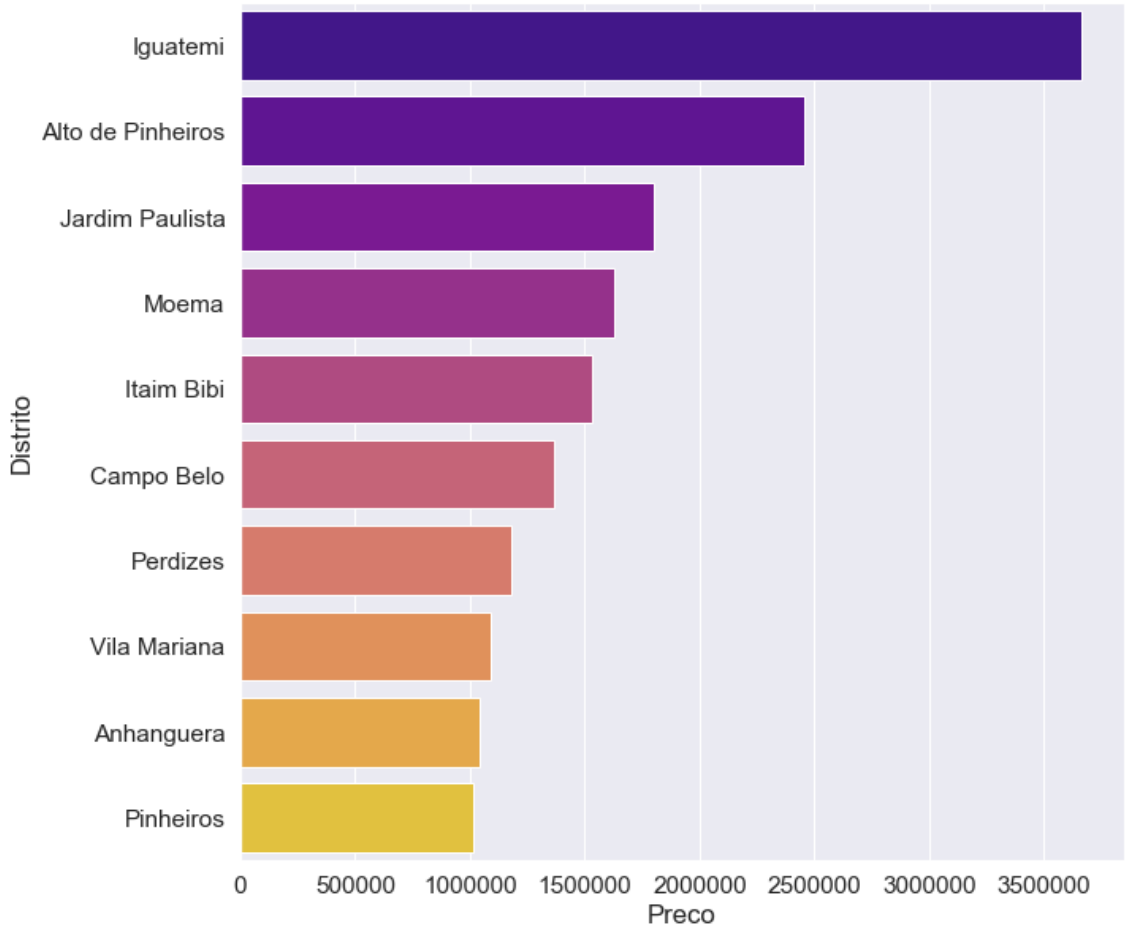


Distritos com os maiores preço médio de apartamentos:



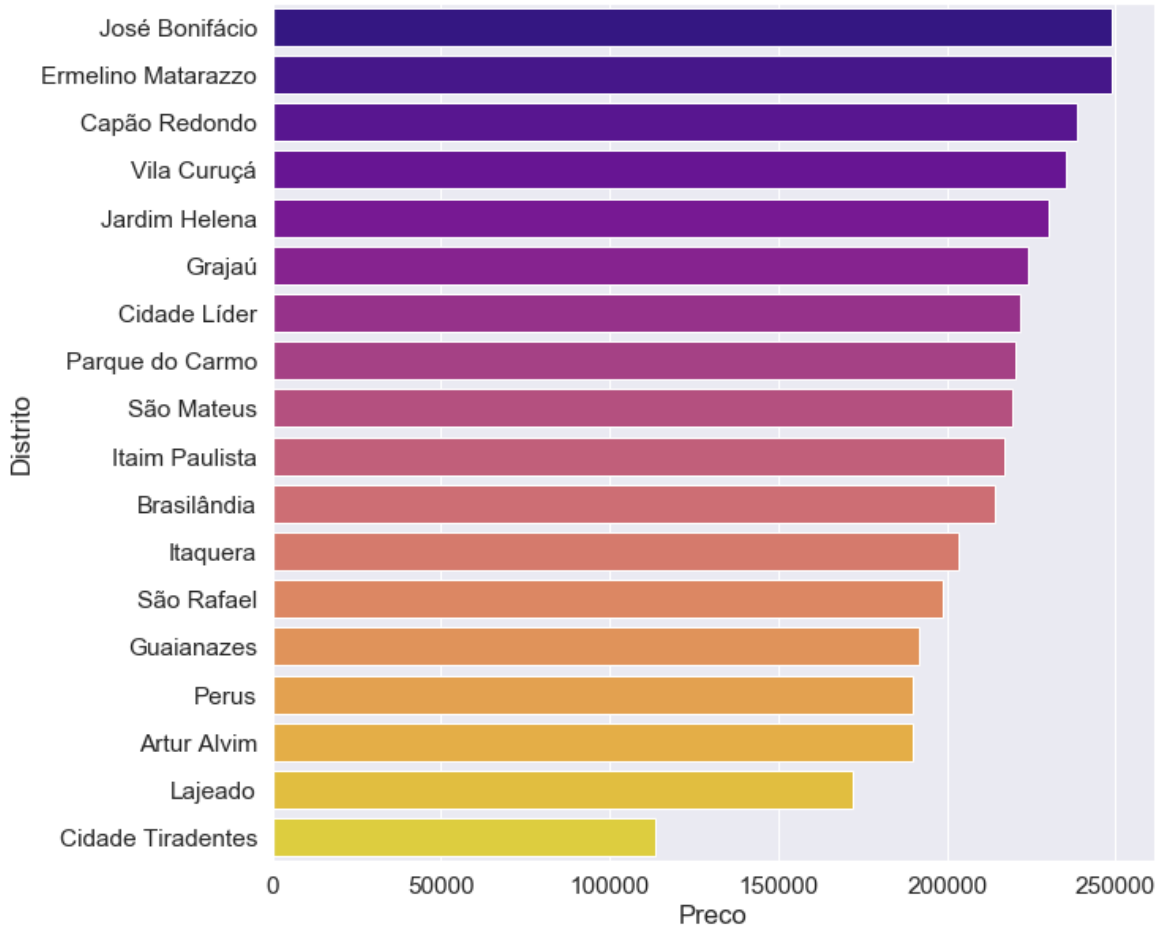
Por fim, segue visualização da distribuição dos dados por Distritos. Previamente se verifica que em 10 distritos o preço médio dos apartamentos colocados à venda supera o valor de 1 milhão de reais.

```
Total de distritos com preço médio acima de 1 milhão de reais:  10
```



Também se verificou um total de 18 distritos em que a média de preço de venda é inferior a duzentos e cinquenta mil reais.

```
Total de distritos com preço médio abaixo de 250 mil de reais:  18
```



Com a biblioteca WordCloud é possível gerar uma nuvem de palavras (*Wordcloud*) a qual permite uma visualização simples e rápida dos dados que aparecem com maior frequência em uma coluna do tipo *object*. No caso permite visualizar com mais destaque quais os Distritos possuem mais imóveis à venda.

```
# Dicionário para gerar Nuvem de Palavras (worldcloud)
data_cloud = dict(df['Distrito'].value_counts())
# gerar uma wordcloud
wordcloud = WordCloud(background_color="white", colormap = "Greens_r",
                      width=1600, height=1000, max_words=93,
                      max_font_size=250,
                      min_font_size=3).generate_from_frequencies(data_cloud)
# mostrar a imagem final
fig, ax = plt.subplots(figsize=(14,14))
ax.imshow(wordcloud, interpolation='bilinear')
ax.set_axis_off()

plt.imshow(wordcloud);
```

Abaixo segue a nuvem de palavras que evidencia os distritos com mais imóveis à venda. Há a predominância de anúncios dos distritos de Itaim Bibi, Bom Retiro, Bela Vista e Casa Verde, seguidos de Moema e Brás.

4.1 Tratamento de dados nulos, faltantes ou incorretos

```
# A princípio não há dados nulos

print('Total de dados nulos:',df.isna().sum().sum())
```

```
Total de dados nulos: 0
```

A princípio não há dados nulos e nem faltantes, mas já foi verificado que existem valores zero para condomínio que devem ser considerados como faltantes. Também há registros de longitude e latitude incorretos e poderiam ser considerados como outliers, mas por justamente não estarem dentro dos limites da cidade de São Paulo serão considerados como nulos para fins de correção e ajuste. A seguir, segue o tratamento para esses valores.

4.1.1 Tratamento de dados variável “Condo”

Constata-se o total de 1289 registros iguais a zero, também há 15 registros com valores igual ou abaixo de 10 (existência de valores iguais a 1) que também serão considerados como faltantes, conforme figura abaixo usando o método len e a função .loc() do pandas.

```
# Visualizando valores muito baixos, há valores 0 para condomínio.
# E valores muito próximos de Zero

cond_0 = len(df[df['Condo']==0])

prox_0 = len(df[df['Condo']<=10].loc[df['Condo']>0])

print('Total de registros com Condomínio igual a zero: ', cond_0)

print('Total de registros com Condomínio próximo a zero: ', prox_0)
```

```
Total de registros com Condomínio igual a zero:  1289
Total de registros com Condomínio próximo a zero:  15
```

A seguir aplica-se a função condicional .where() da biblioteca numpy para substituir os valores da coluna “Condo” iguais ou inferiores a 10 por “Nan”. Em seguida é usada a função .fillna() e .median() para substituir os registros nulos pela mediana, dado que a coluna “Condo” possui uma distribuição assimétrica, a mediana seria mais representativa do que a média

```
# Substituindo valores iguais ou menores que 10 por None:

df['Condo'] = np.where(df['Condo']<= 10,np.nan, df['Condo'])

# substituir o missing pela mediana da coluna
df['Condo'].fillna(df['Condo'].median(), inplace=True)

df.Condo.describe()
```

```
count    6302.000000
mean      645.827039
std       567.440414
min        20.000000
25%       380.000000
50%       500.000000
75%       700.000000
max      8920.000000
Name: Condo, dtype: float64
```

4.1.2 Tratamento dos dados geográficos: “Latitude” e “Longitude”

A seguir serão trabalhados os registros de latitude e longitude. Primeiro será feita a contagem de registros iguais a zero nessas colunas usando *len() e .loc()*. Foi calculado um total de 394 registros iguais a zero tanto para Latitude quanto Longitude.

```
# Verificando valores iguais a zero:
lat_zero = len(df[['Distrito','Latitude', 'Longitude']].loc[df['Latitude']==0])
long_zero = len(df[['Distrito','Latitude', 'Longitude']].loc[df['Longitude']==0])

print('Total de registros de Latitude iguais a zero: ',lat_zero)
print('Total de registros de Longitude iguais a zero: ',long_zero)
```

```
Total de registros de Latitude iguais a zero:  394
Total de registros de Longitude iguais a zero:  394
```

Segue aplicação da função *.geocode()* da biblioteca “Nominatim”, que combinda com as regras de busca irá retornar um dicionário contendo as latitudes e longitudes limites da cidade de São Paulo. De forma análoga ao tratamento dos valores de condomínio, posteri-

ormente, esses valores serão usados para substituir as latitudes que não estejam dentro do intervalo mínimo e máximo por 'Nan', com o uso da função *.where()* e *.loc()*.

```
# Uso Biblioteca Nominatim para identificar a Latitude e Longitude Máxima e Mínima da cidade de São Paulo
place = 'São Paulo, Região Imediata de São Paulo, Região Metropolitana de São Paulo,'

geolocator = Nominatim(user_agent="geolocalização")
d_lat_sp={}
d_long_sp={}
location = geolocator.geocode( place)
d_lat_sp['São Paulo']=[float(location.raw['boundingbox'][0]),float(location.raw['boundingbox'][1])]
d_long_sp['São Paulo']= [float(location.raw['boundingbox'][2]), float(location.raw['boundingbox'][3])]

print('Latitudes Mínima e Máxima:', d_lat_sp)
print('Longitudes Mínima e Máxima:',d_long_sp )
```

```
Latitudes Mínima e Máxima: {'São Paulo': [-23.7106507, -23.3906507]}
Longitudes Mínima e Máxima: {'São Paulo': [-46.7933824, -46.4733824]}
```

```
# Substituindo os registros de posição geográfica incorretos por NaN(fora dos limites máximos e mínimos)

df['Latitude'] = np.where(df['Latitude']>=d_lat_sp['São Paulo'][0],
                          np.where(df['Latitude']<=d_lat_sp['São Paulo'][1],
                          df['Latitude'],
                          np.nan),np.nan)

df['Longitude'] = np.where(df['Longitude']>=d_long_sp['São Paulo'][0],
                           np.where(df['Longitude']<=d_long_sp['São Paulo'][1],
                           df['Longitude'],
                           np.nan),np.nan)
```

A partir do dataframe principal, são criados outros dois dataframes contendo as médias das latitudes e longitudes para cada distrito usando as funções *.groupby(), .mean()* e *.reset_index()* do pandas. Após cria-se um novo dataframe ("df_medias") a partir da junção (*inner join*) dos registros que são comuns a esses dois novos dataframes usando a função .merge(). Aplica-se *.dropna()* para eliminar eventuais valores nulos. Assim obtêm-se a média de latitudes e longitudes em cada distrito para substituir os registros nulos:

```
# A partir do df, cria outros dois dataframes com as médias das latitudes e longitudes para cada distrito.
# Com merge cria-se novo df (df_medias) com as medias e elimina-se os valor nulos
# Assim obtêm-se a média de latitudes e longitudes em cada distrito para substituir os registros nulos:

df_media_lat = df[['Distrito', 'Latitude']].groupby(['Distrito']).mean().reset_index()
df_media_long = df[['Distrito', 'Longitude']].groupby(['Distrito']).mean().reset_index()

df_medias = pd.merge(df_media_lat, df_media_long, on = 'Distrito', how = 'inner')
df_medias.dropna(inplace=True)
print('Dimensão do DataFrame df_medias: ', df_medias.shape)
display(df_medias.head())
```

```
Dimensão do DataFrame df_medias:  (88, 3)
```

	Distrito	Latitude	Longitude
0	Alto de Pinheiros	-23.546908	-46.710279
1	Anhanguera	-23.665496	-46.691405
2	Aricanduva	-23.557348	-46.523902
3	Artur Alvim	-23.544183	-46.484449
4	Barra Funda	-23.523643	-46.663635

Detalhe importante é que ainda falta ajustar as latitudes e longitudes de 5 distritos. Aplicando o método *.unique(), set(), list()* e o operador de subtração (-), obtem-se quais distritos ainda não têm dados geográficos.

```
# Verificando quais os distritos não contêm posição geográfica subtraindo-os do df_medias no df,
# resultando numa relação de distritos sem as médias das posições geográficas.

places = list(set(df['Distrito'].unique())-set(df_medias['Distrito'].unique()))
print( "Relação de Distritos sem valores de Latitude e Longitude:\n\n", places)
```

```
Relação de Distritos sem valores de Latitude e Longitude:

 ['Grajaú', 'Vila Jacuí', 'Itaquera', 'Cidade Tiradentes', 'Lajeado']
```

Novamente será usada a função *.geocode()* que irá retornar o valor de latitude e longitude dos distritos faltantes. Adicionando esses valores no dataframe “df_medias”. Por fim, visualiza-se os últimos valores desse dataframe.

```
# Com a biblioteca Nominatim, encontra-se Latitude e Longitude e insere no df_medias:

places = list(set(df['Distrito'].unique())-set(df_medias['Distrito'].unique()))

geolocator = Nominatim(user_agent="geolocalização")
for place in places:
    location = geolocator.geocode( place+'- São Paulo - SP')
    df_medias = df_medias.append({'Distrito' : place , 'Latitude' : location.latitude,
                                  'Longitude' : location.longitude}, ignore_index=True)

df_medias.tail()
```

	Distrito	Latitude	Longitude
88	Grajaú	-23.779971	-46.673766
89	Vila Jacuí	-23.500294	-46.458717
90	Itaquera	-23.536080	-46.455510
91	Cidade Tiradentes	-23.582497	-46.409207
92	Lajeado	-23.536248	-46.410022

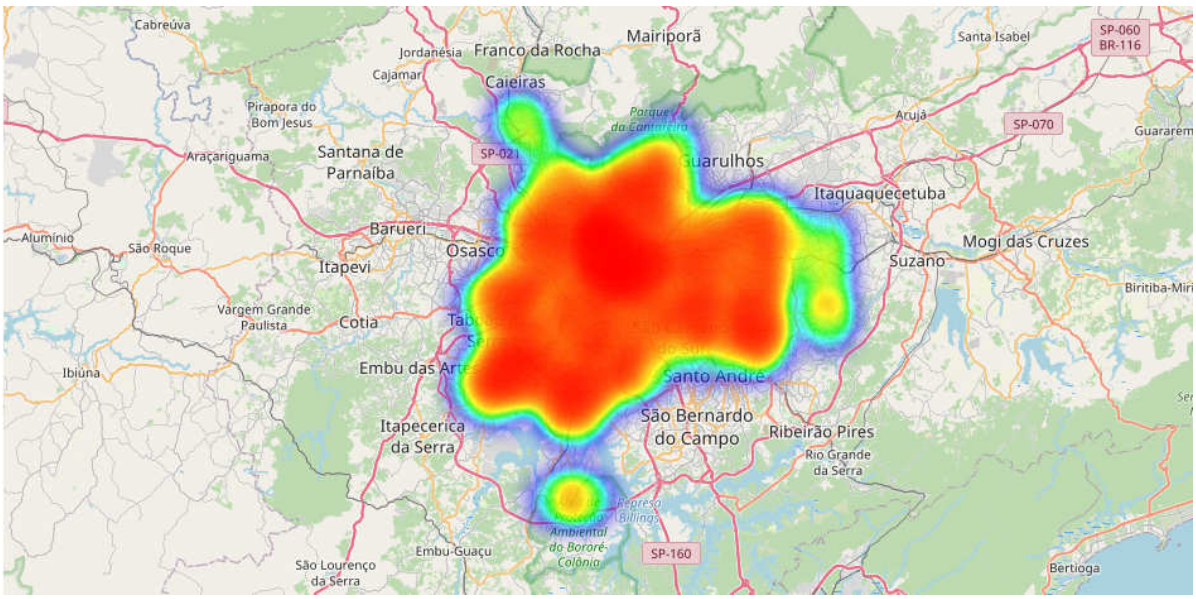
A fim de facilitar a manipulação dos dataframes, as colunas do df_medias foram renomeadas para ‘media_Latitude’ e ’media_Longitude’. Finalmente podem ser substituídos os valores incorretos com a a junção (*left join*) dos registros que estão no “df” e dos registros que são comuns ao “df_medias”. A seguir aplica-se *.fillna()* nas colunas Latitude e Longitude preenchendo os valores nulos com os valores das colunas ‘media_Latitude’ e ’media_Longitude’. Conclui-se eliminando essas colunas advindas do “df_medias” e retornando a contagem total dos registros de Latitude e Longitude com a função *.count().*

```
# No dataframe df, substitui as latitudes e longitudes incorretas pelo nome do respectivo distrito
df_medias.rename(columns={'Latitude':'media_Latitude', 'Longitude': 'media_Longitude'}, inplace=True)
df = pd.merge(df, df_medias, on='Distrito', how='left')

df["Latitude"].fillna(df['media_Latitude'], inplace=True)
df["Longitude"].fillna(df['media_Longitude'], inplace=True)

df.drop(columns = ['media_Latitude', 'media_Longitude'], inplace = True)
print('Total registros de Latitude:', df.Latitude.count())
print('Total registros de Longitude:', df.Longitude.count())
```

```
Total registros de Latitude: 6302
Total registros de Longitude: 6302
```

Após esse tratamento, a distribuição geográfica dos imóveis concentra a totalidade dos registros em São Paulo, conforme demonstram o mapa abaixo:

Com a mesma biblioteca do folium, criou-se mapa de calor que permite visualizar melhor a distribuição dos imóveis na área urbana da cidade.



4.2 *Feature Engineering*

Após exploração dos dados e tratamento dos dados nulos é feita a checagem de dados duplicados e respectiva eliminação e existência de dados nulos usando *.drop_duplicates()* e *.dropna()*. e informando a atual dimensão do dataframe. A seguir será realizada a etapa engenharia de atributos (*feature enginering*) dos dados para otimizar a informação contida no dataframe.

```
df.dropna(inplace=True)
df = df.drop_duplicates()

df.reset_index(drop = True,inplace=True)
display(df.round().describe())
#display(df.describe())
print('Total de valores NaN: ', df.isna().sum().sum())
print('Dimensão do Dataframe: ', df.shape)
```

	Preco	Condo	Area	Quartos	Banheiros	Suites	Garagem	Elevador	Mobiliado	Piscina	...	Densidade Demográfica (Hab/km²)
count	6301.00	6301.00	6301.00	6301.00	6301.00	6301.00	6301.00	6301.00	6301.00	6301.00	...	6301.00
mean	613247.70	645.91	79.01	2.33	2.04	0.94	1.33	0.42	0.12	0.54	...	11817.80
std	744926.40	567.45	51.08	0.71	0.92	0.77	0.76	0.49	0.32	0.50	...	5192.49
min	42000.00	20.00	30.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	...	1978.00
25%	250000.00	380.00	50.00	2.00	2.00	1.00	1.00	0.00	0.00	0.00	...	8361.00
50%	381000.00	500.00	62.00	2.00	2.00	1.00	1.00	0.00	0.00	1.00	...	11104.00
75%	680000.00	700.00	88.00	3.00	2.00	1.00	2.00	1.00	0.00	1.00	...	14671.00
max	10000000.00	8920.00	620.00	6.00	7.00	6.00	7.00	1.00	1.00	1.00	...	26715.00

8 rows × 25 columns

```
Total de valores NaN:  0
Dimensão do Dataframe:  (6301, 26)
```

4.2.1 Conversão de dados categóricos

A seguir será feita a conversão das variáveis do tipo ‘object’ para numérico (*int8*). Para tanto, foi aplicado recurso *for in* e condicional para compor lista contendo nome das colunas do tipo objeto e numérica. A seguir é aplicado método *.Categorical* e *cat.codes* do pandas que converterá os dados categóricos em numéricos.

```
# Convertendo variáveis do tipo object para inteiro
## chama função para relacionar as colunas numéricas e categóricas
col = df.columns
col_object=[]
col_numeric=[]
for i in col:
    if df[i].dtypes in [np.object]:
        col_object.append(i)
    elif df[i].dtypes in [np.int64, np.float64]:
        col_numeric.append(i)

print('Variáveis tipo object: ', col_object)

for col in col_object:
    df[col] = pd.Categorical(df[col])
    df[col] = df[col].cat.codes

df.Distrito.describe()
```

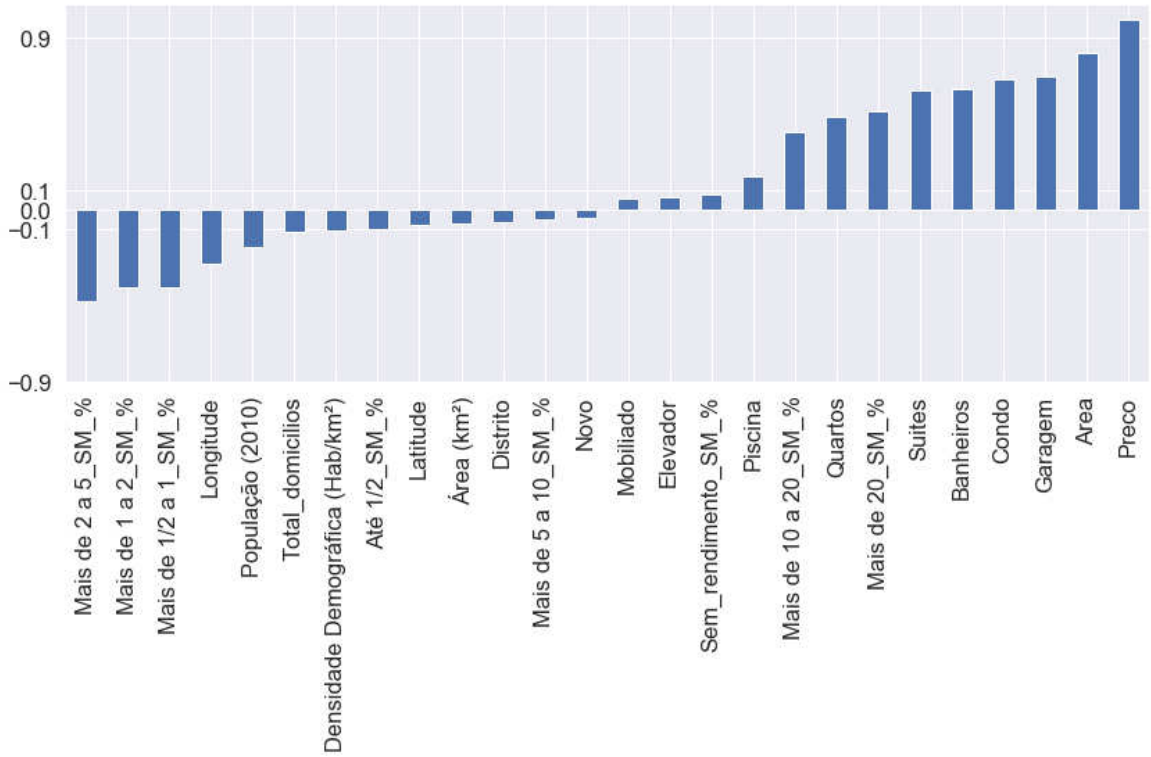
```
Variáveis tipo object:  ['Distrito']

count   6301.00
mean      43.08
std       27.23
min        0.00
25%       19.00
50%       42.00
75%       66.00
max       92.00
Name: Distrito, dtype: float64
```

4.2.2 Baixa correlação

A seguir será analisada a correlação da variável “Preco” com as demais variáveis que irão alimentar o modelo de *machine learning*. Para tanto, será usada a função *.corr()* que tem por padrão usar o coeficiente de correlação de Pearson, que é um teste que mede a relação estatística entre duas variáveis. Valores positivos próximos de +1 indicam forte correlação positiva e valores negativos próximos de -1 indicam forte correlação negativa. Valores próximos de zero indicam que a correlação é muito fraca. Abaixo segue o gráfico com os coeficientes de correlação; foi marcado os intervalos entre [-0,1 e +0,1].

```
#A variável Preco e a correlação com as demais variáveis

df.corr()['Preco'].sort_values().plot(kind = 'bar',yticks = [-.9,-.1,0,.1,.9], mark_right=False, figsize=(14,5))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x19a9f0b9b70>
```



Percebe-se quais as variáveis apresentam coeficientes abaixo de |0.1|, sendo comum na literatura que coeficientes de correlação abaixo desse valor sejam considerados desprezíveis (Callegari,2009). A seguir será criada lista que seleciona as colunas com coeficiente maior que |0.1|.

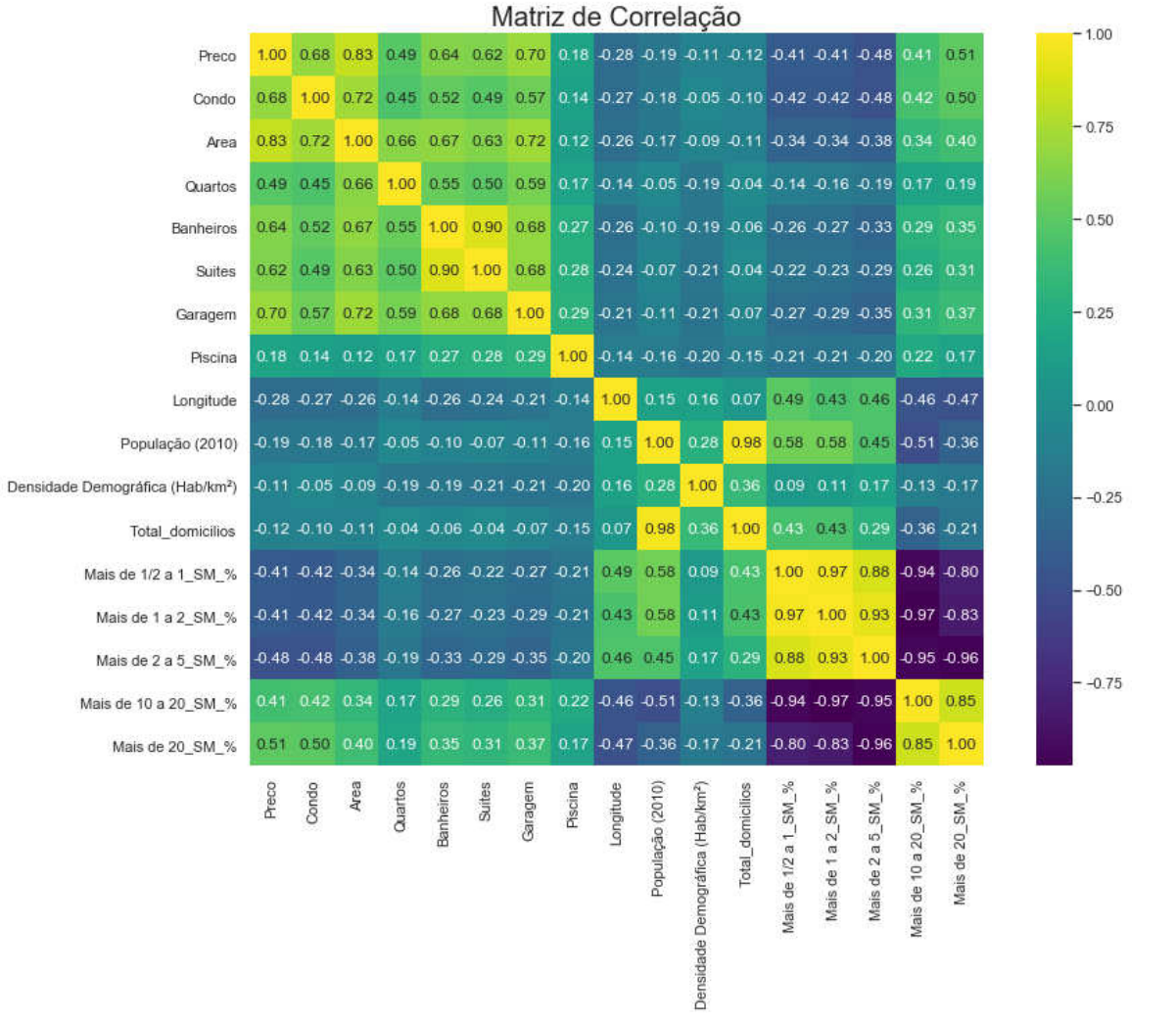
```
# Selecionando as variáveis com correlação absoluta maior que |0.1|

s_corr = df.corr()['Preco'].abs()

s_corr = s_corr.loc[s_corr>0.1]
s_corr.sort_values(ascending=False)
l_select_cols = list(s_corr.index)

print('Variáveis consideradas para verificar multicolinearidade:', l_select_cols)
```

```
Variáveis consideradas para verificar multicolinearidade: ['Preco', 'Condo', 'Area', 'Quartos', 'Banheiros', 'Suites', 'Garage
m', 'Piscina', 'Longitude', 'População (2010)', 'Densidade Demográfica (Hab/km²)', 'Total_domicilios', 'Mais de 1/2 a 1_SM_%',
'Mais de 1 a 2_SM_%', 'Mais de 2 a 5_SM_%', 'Mais de 10 a 20_SM_%', 'Mais de 20_SM_%']
```

4.2.3 Multicolinearidade

Outro problema que afetam o desenvolvimento dos modelos de aprendizado de máquina é a multicolinearidade das variáveis preditoras, problema no qual essas variáveis possuem relações lineares exatas ou aproximadamente exatas. É um indicativo da existência de multicolinearidade um valor de correlação elevado. Plotou-se mapa de calor com os coeficientes de correlação das variáveis previamente selecionadas na etapa anterior. Para isso, foi usada a função .heatmap() do seaborn e 'veridis' como parâmetro de coloração de modo que evidencie os valores mais extremos.

```
# Plotando o mapa de correlação entre as variáveis
sns.set(font_scale = 1)
plt.figure(figsize=(14,14))
plt.title("Matriz de Correlação", fontsize=20)
sns.heatmap(df[l_select_cols].corr(), cbar=True, square= True, fmt='.2f', annot=True, cmap='viridis')
```



Um coeficiente de correlação acima de 0.9 é classificado como muito forte (Callegari,2009). Para seleção das variáveis a serem descartadas, num primeiro momento, considera-se aquelas com um coeficiente igual ou superior a |0.90| entre as variáveis preditoras. A seguir, dentre essas variáveis selecionadas (igual ou acima de |0,90|), é descartada a que tiver menor coeficiente de correlação com a variável resposta ("Preco"). A título de exemplo, foi verificado que "Banheiros" e "Suites" são fortemente relacionados com coeficiente igual a 0,90, então foi descartada a variável "Suites" porque seu coeficiente de correlação com a variável "Preco" é menor que o da variável "Banheiros".

Segue método que seleciona as variáveis com coeficiente de correlação maior que 0.9 entre si e que retorna um dataframe que lista as variáveis e seu coeficiente de correlação com a variável "Preco". Para isso, foi criado um dataframe (df_corr), cujo formato é análogo ao do mapa de calor. Abaixo seguem os primeiros registros mostrados do dataframe.

```
# Multicolinearidade - verificando quais as variáveis preditoras estão correlacionadas

df_corr = df[l_select_cols].corr()
df_corr = pd.DataFrame(df_corr)
df_corr.head()
```

	Preco	Condo	Area	Quartos	Banheiros	Suites	Garagem	Piscina	Longitude	População (2010)	Densidade Demográfica (Hab/km²)	T
Preco	1.00	0.68	0.83	0.49	0.64	0.62	0.70	0.18	-0.28	-0.19	-0.11	
Condo	0.68	1.00	0.72	0.45	0.52	0.49	0.57	0.14	-0.27	-0.18	-0.05	
Area	0.83	0.72	1.00	0.66	0.67	0.63	0.72	0.12	-0.26	-0.17	-0.09	
Quartos	0.49	0.45	0.66	1.00	0.55	0.50	0.59	0.17	-0.14	-0.05	-0.19	
Banheiros	0.64	0.52	0.67	0.55	1.00	0.90	0.68	0.27	-0.26	-0.10	-0.19	

Na sequência é criada nova coluna com o respectivo index a fim de aplicar a função *.melt()* para unificar os valores dessa matriz. Assim será reformulado o dataframe filtrando por valor os coeficientes maiores ou iguais a |0.90|. A partir desse dataframe, formará lista com os valores únicos da coluna "índice", a partir da qual será gerado dataframe com os coeficientes de correlação em relação a variável "Preco". Esse novo dataframe ("df_corr_preco") será usado para selecionar as variáveis a serem desprezadas.

```
# Seleção das variáveis preditoras correlacionadas e criação do dataframe com a respectiva correlação com Preco
df_corr['indice'] = df_corr.index
df_corr = df_corr.melt(id_vars='indice')
df_corr = df_corr.sort_values(by = 'indice',
                              ascending=False)[(abs(round(df_corr['value'],
                                                          2))>=.90)].loc[df_corr['indice']!= df_corr['variable']]

columns = list(df_corr['indice'].unique())
columns.append('Preco')
df_corr_preco = pd.DataFrame(df[columns].corr()['Preco'].abs()).reset_index().sort_values(by='Preco',ascending=False)
df_corr_preco
```

	index	Preco
9	Preco	1.00
8	Banheiros	0.64
1	Suites	0.62
3	Mais de 20_SM_%	0.51
4	Mais de 2 a 5_SM_%	0.48
7	Mais de 1 a 2_SM_%	0.41
5	Mais de 10 a 20_SM_%	0.41
6	Mais de 1/2 a 1_SM_%	0.41
2	População (2010)	0.19
0	Total_domicilios	0.12

Selecionando as variáveis para descarte. Adiciona-se os dados desses dois dataframes (df_corr e df_corr_preco) através de *left join* usando *.merge()*. Foram adicionadas mais 4 colunas, a primeira (index_x) contendo a variável constante na coluna ‘indice’, a segunda coluna (Preco_x) informando o respectivo coeficiente de correlação com a variável “Preco”. Já a terceira coluna (index_y), contendo nome da variável da coluna ‘variable’, sendo a quarta coluna (Preco_y) informando o respectivo coeficiente de correlação com a variável “Preco”.

```
# Dentre essas variáveis, serão excluídas aquelas de menor grau de correlação em relação a variável Preco.
# Com a função .merge() serão acrescentadas colunas referentes a variável e sua correlação com a preco
df_corr = pd.merge(df_corr, df_corr_preco, how = 'left', left_on='indice', right_on = 'index')
df_corr = pd.merge(df_corr, df_corr_preco, how = 'left', left_on='variable', right_on = 'index')
df_corr.head()
```

	indice	variable	value	index_x	Preco_x	index_y	Preco_y
0	Total_domicilios	População (2010)	0.98	Total_domicilios	0.12	População (2010)	0.19
1	Suites	Banheiros	0.90	Suites	0.62	Banheiros	0.64
2	População (2010)	Total_domicilios	0.98	População (2010)	0.19	Total_domicilios	0.12
3	Mais de 20_SM_%	Mais de 2 a 5_SM_%	-0.96	Mais de 20_SM_%	0.51	Mais de 2 a 5_SM_%	0.48
4	Mais de 2 a 5_SM_%	Mais de 1 a 2_SM_%	0.93	Mais de 2 a 5_SM_%	0.48	Mais de 1 a 2_SM_%	0.41

A seguir são criadas mais duas colunas a partir da função condicional .where() do numpy. A condição da coluna ‘manter’ retornará qual das variáveis contidas em ‘index_x’ e ‘index_y’ possui maior coeficiente de correlação com o preço. Já a coluna ‘excluir’ informará qual possui menor coeficiente de correlação com preço.

```
# Agora são criadas colunas que irão selecionar quais as que tem maior correlação e as de menor correlação
df_corr["manter"] = np.where(df_corr['Preco_x']>df_corr['Preco_y'],df_corr['index_x'], df_corr['index_y'])
df_corr["excluir"] = np.where(df_corr['Preco_x']<df_corr['Preco_y'],df_corr['index_x'], df_corr['index_y'])
df_corr.head()
```

	indice	variable	value	index_x	Preco_x	index_y	Preco_y	manter	excluir
0	Total_domicilios	População (2010)	0.98	Total_domicilios	0.12	População (2010)	0.19	População (2010)	Total_domicilios
1	Suites	Banheiros	0.90	Suites	0.62	Banheiros	0.64	Banheiros	Suites
2	População (2010)	Total_domicilios	0.98	População (2010)	0.19	Total_domicilios	0.12	População (2010)	Total_domicilios
3	Mais de 20_SM_%	Mais de 2 a 5_SM_%	-0.96	Mais de 20_SM_%	0.51	Mais de 2 a 5_SM_%	0.48	Mais de 20_SM_%	Mais de 2 a 5_SM_%
4	Mais de 2 a 5_SM_%	Mais de 1 a 2_SM_%	0.93	Mais de 2 a 5_SM_%	0.48	Mais de 1 a 2_SM_%	0.41	Mais de 2 a 5_SM_%	Mais de 1 a 2_SM_%

A partir dessas colunas são criadas listas, com as quais são obtidas as variáveis que devem ser descartadas. Para isso, realiza-se operações de conjunto. Usando métodos .unique() e set() nas colunas ‘manter’ e ‘excluir’ é possível aplicar o operador de subtração (-) e converter o resultado em lista.

Por fim, a lista *l_select_cols* será resultado dela mesma subtraída da lista que contêm as colunas a serem excluídas.

```
# A seguir é aplicado método para listar quais as variáveis que devem ser mantidas e quais devem ser excluídas
# as variáveis a serem excluidas serão subtraídas da lista de variáveis que tinham correlação maior que 0.1
l_manter = list(set(df_corr["manter"].unique())-set(df_corr["excluir"].unique()))
l_excluir = list(set(df_corr["excluir"].unique())-set(l_manter))
l_select_cols = list(set(l_select_cols)-set(l_excluir))

print('Colunas mantidas:', l_select_cols,'\n\nColunas a serem excluídas por Multicolinearidade:',l_excluir)
```

```
Colunas mantidas: ['Quartos', 'Densidade Demográfica (Hab/km²)', 'Condo', 'Area', 'Mais de 20_SM_%', 'População (2010)', 'Pisci
na', 'Banheiros', 'Garagem', 'Longitude', 'Preco']

Colunas a serem excluídas por Multicolinearidade: ['Suites', 'Mais de 2 a 5_SM_%', 'Mais de 10 a 20_SM_%', 'Mais de 1 a 2_SM
_%', 'Mais de 1/2 a 1_SM_%', 'Total_domicilios']
```

Plotando mapa de calor usando *.heatmap()* com as variáveis selecionadas é possível verificar que já não há nenhum coeficiente de correlação maior que |0.90| e na linha da variável “Preco” todos os coeficientes são maiores que |0.1|

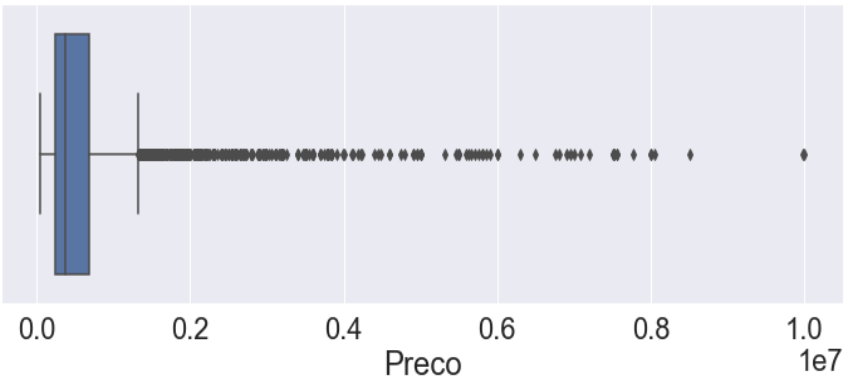
Após essa checagem, será feita a seleção das colunas de interesse.

```
# Selecionando variáveis da lista l_select_cols:

df=df[l_select_cols]
```

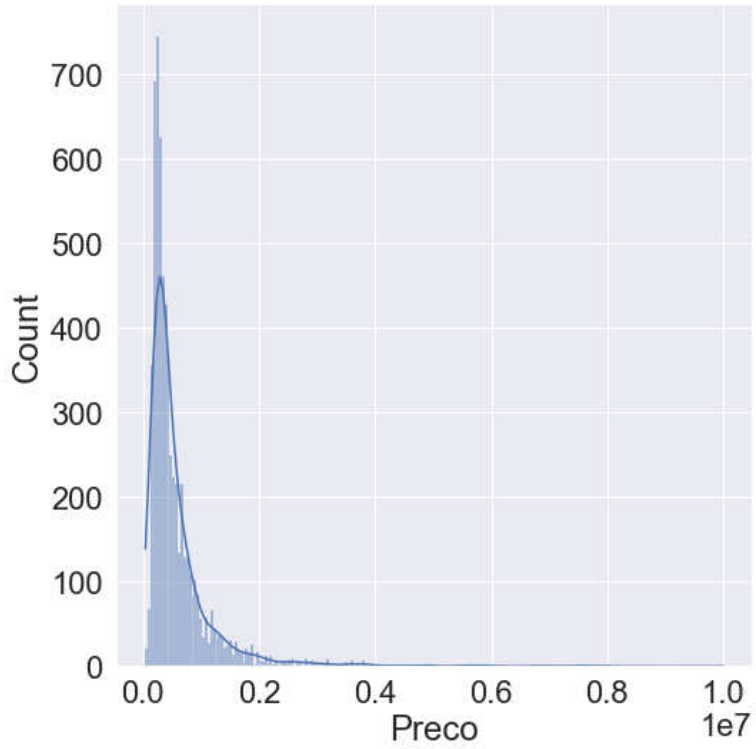
4.2.4 Análise variável resposta (*target*)

Visualização da variável Preco através da função *.describe()* já evidência que a distribuição não é normal, que a média é quase o dobro da mediana, enquanto que o preço mínimo é de 42 mil e o valor máximo é 10 milhões (grande amplitude). Aplicando a função *.boxplot(),* pode-se ver um número bastante elevado de outliers à direita e com a função *.displot(),* evidencia-se a assimetria da distribuição desses valores. A seguir será tratado esse caso da assimetria dos dados.

```
AxesSubplot(0.125,0.125;0.775x0.755)

count        6301.0
mean       613247.7
std        744926.4
min         42000.0
25%        250000.0
50%        381000.0
75%        680000.0
max      10000000.0
Name: Preco, dtype: object
```



<Figure size 864x288 with 0 Axes>



4.2.5 Transformação de variáveis contínuas assimétricas

Para harmonizar a distribuição de frequências, serão convertidas em logaritmo natural (ln) as variáveis que possuem distribuição assimétrica. Para aferir o grau de assimetria de cada distribuição será usada a função *.skewness()* em cada coluna numérica. A função .skewness() é uma medida de assimetria da distribuição, sendo que o valor de assimetria deve ser aproximadamente zero para dados normalmente distribuídos. Quando o valor da assimetria é negativo, a cauda da distribuição é mais longa em direção ao lado esquerdo da curva (distribuição assimétrica à esquerda ou negativa). Já quando o valor da assimetria é positivo, a cauda da distribuição é mais longa em direção ao lado direito da curva (distribuição assimétrica à direita ou positiva).

A função *simetric(df, colunas)* retorna colunas convertidas para logaritimo natural usando a medida de assimetria (skewness) como critério. Se for maior que 1 ou menor que -1, ela transformará os valores da coluna para logaritmo natural e calculará a assimetria des-

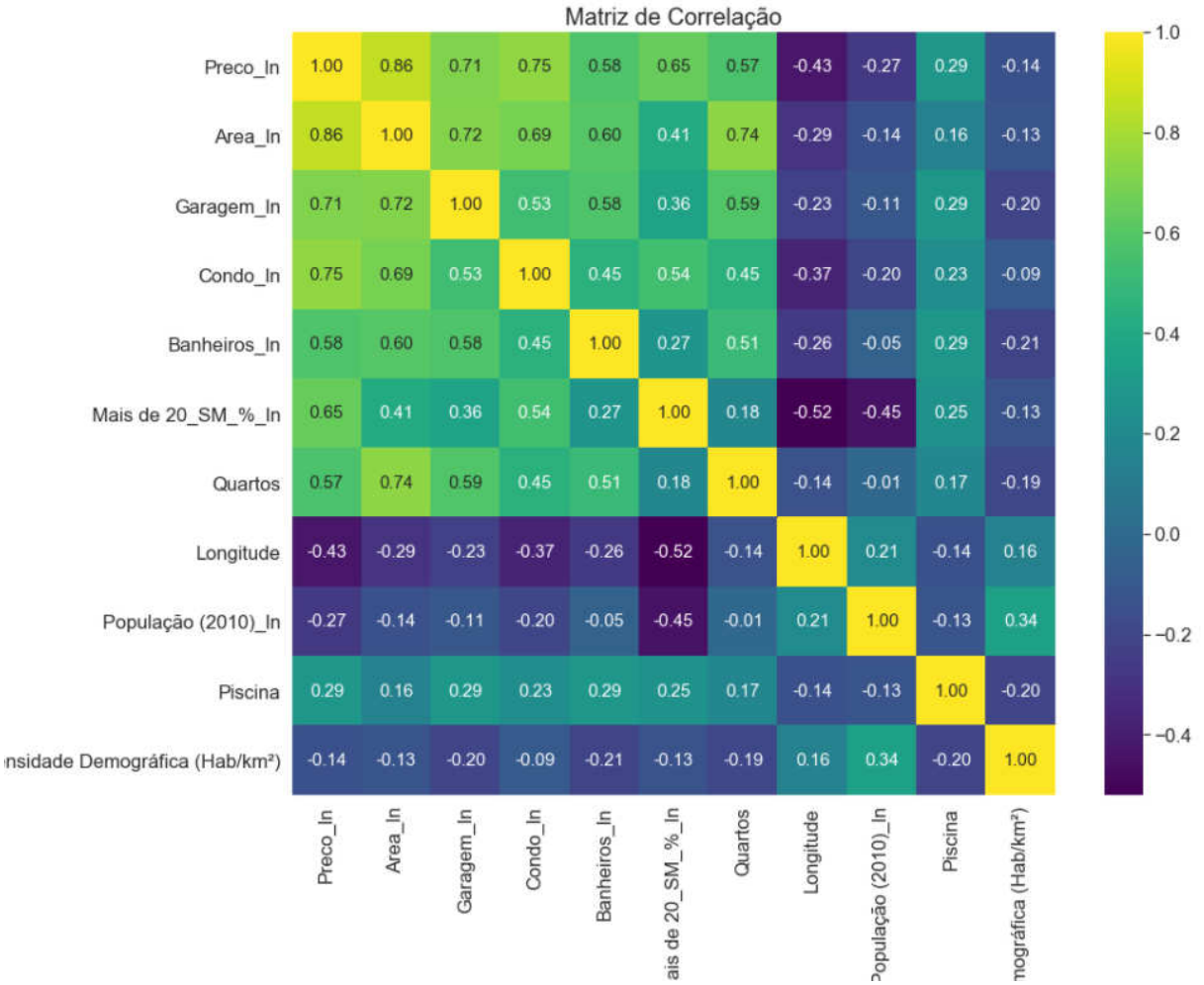
ses dados transformados. Caso a nova coluna tenha um valor de assimetria menor, a coluna original é excluída, caso contrário, a coluna original será mantida. Como resultado a função imprime os respectivos valores de assimetria para cada coluna e informa se foi ajustada ou não.

```
# Criando novo dataframe com base logatmica para variáveis com distribuição assimétrica

df_ln = df.copy()
cols_ln = simetric(df_ln, l_select_cols)
```

```
Assimetria (Skewness) em Preco é 5.08
Assimetria (Skewness) em Preco_ln é 0.84
Coluna Preco ajustada

Assimetria (Skewness) em Area é 2.88
Assimetria (Skewness) em Area_ln é 1.09
Coluna Area ajustada

Assimetria (Skewness) em Garagem é 1.82
Assimetria (Skewness) em Garagem_ln é 1.44
Coluna Garagem ajustada

Assimetria (Skewness) em Condo é 4.17
Assimetria (Skewness) em Condo_ln é 0.11
Coluna Condo ajustada

Assimetria (Skewness) em Banheiros é 1.65
Assimetria (Skewness) em Banheiros_ln é 0.11
Coluna Banheiros ajustada

Assimetria (Skewness) em Mais de 20_SM_% é 1.24
Assimetria (Skewness) em Mais de 20_SM_%_ln é -0.42
Coluna Mais de 20_SM_% ajustada

Assimetria (Skewness) em População (2010) é 1.34
Assimetria (Skewness) em População (2010)_ln é -0.58
Coluna População (2010) ajustada
```

Antes de aplicar a função *simetric(df, cols),* foi criada cópia do dataframe original (df_ln) a qual vai ser submetida para conversão logarítmica. Após aplicar a função, verifica-se que das 11 variáveis, 7 delas foram transformadas, inclusive a variável resposta. Plotando o mapa de calor nota-se que os coeficientes de correlação sofreram algumas mudanças. Com relação à variável preço, o coeficiente variou positivamente em todas as variáveis, a princípio indicando um aumento da capacidade explicativa. Também em relação às variáveis preditoras, não há coeficiente acima de |0.90|, havendo baixo impacto na multicolinearidade.

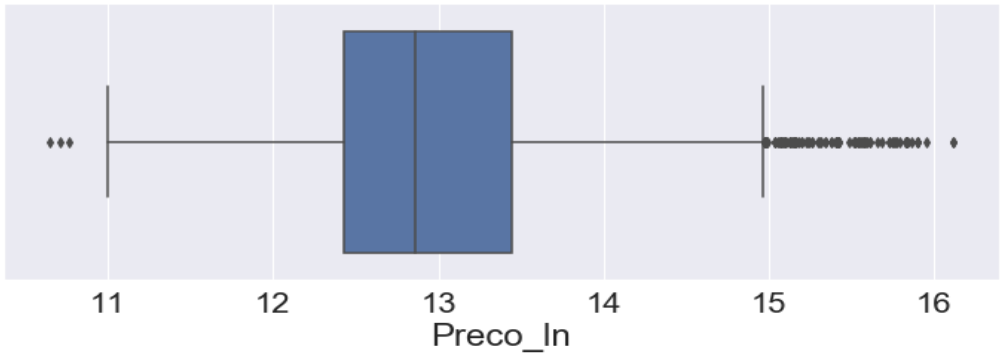
```
# Plotando o mapa de correlação entre as variáveis
df_ln=df_ln[['Preco_ln', 'Area_ln', 'Garagem_ln', 'Condo_ln', 'Banheiros_ln',
       'Mais de 20_SM_%_ln','Quartos', 'Longitude','População (2010)_ln', 'Piscina','Densidade Demográfica (Hab/km²)']]

sns.set(font_scale = 1.5)
plt.figure(figsize=(15,12))
plt.title("Matriz de Correlação", fontsize=20)
sns.heatmap(df_ln.corr(), cbar=True, square= True, fmt='.2f', annot=True, annot_kws={'size':15}, cmap='viridis')
```

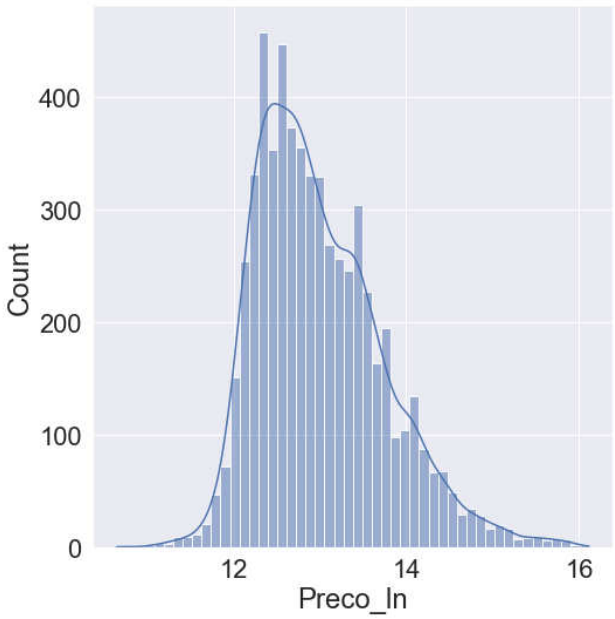


Conforme a descrição, diagrama de caixa e a distribuição de frequência da variável resposta do dataframe “df_ln”, é nítido que houve redução da assimetria e que a transformação para logaritmo natural reduziu o número de possíveis outliers. Para fins de comparação, o dataframe original será mantido para o tratamento de outliers e posterior aplicação

dos modelos de machine learning para fins de comparação de resultados e escolha de qual dataframe ser usado.

```
AxesSubplot(0.125,0.125;0.775x0.755)

count    6043.0
mean       13.0
std         0.75
min        10.65
25%        12.43
50%        12.86
75%        13.44
max        16.12
Name: Preco_ln, dtype: object
```





4.2.6 Tratamento de *outliers*

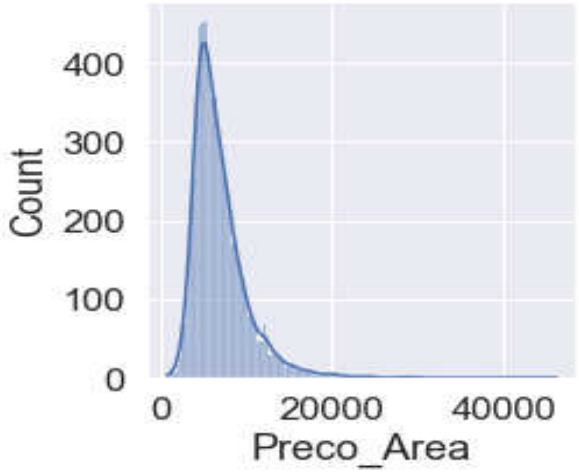
Os outliers são valores que fogem da normalidade e que podem causar anomalias nos resultados obtidos pelos modelos de aprendizado. Como já visto, a variável resposta possui alguns valores discrepantes que apontam a existência de outliers, mas antes de realizar a exclusão ou alteração desses dados, são necessárias duas considerações: primeiro que a variável “Area” é a que tem maior coeficiente de correlação em relação a “Preco”, sendo a variável que mais influencia no preço de venda. Segundo que o valor do metro quadrado, obtido pela divisão do preço do imóvel pela sua área, é um atributo importante para análise de preço no mercado imobiliário

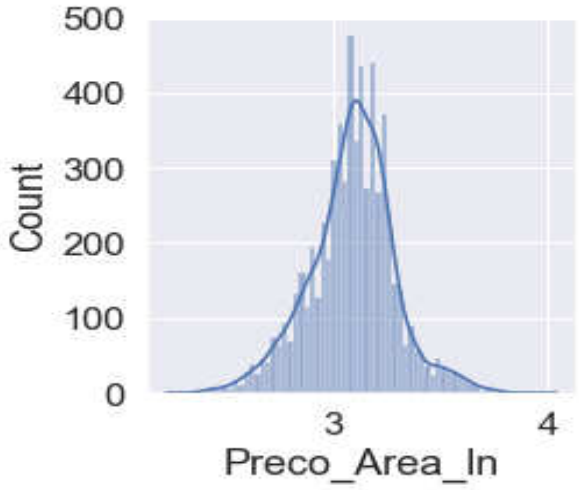
Tendo em vista essas considerações, o preço por metro quadrado seria uma referência mais apropriada para tratar *outliers* da variável resposta, uma vez que se um imóvel tiver tanto o preço quanto área muito elevado, não necessariamente o preço por metro quadrado será também elevado.

“Preco” e “Area” possuem distribuição assimétrica à direita. Para fins de ajuste e harmonização para identificar *outliers*, será ajustado o preço dividindo pela área do imóvel, criando uma nova coluna no dataframe (“Preco_Area”) equivalente a preço por metro quadrado, dessa forma, imóveis de preços elevados justamente porque são muito grandes tenderão a permanecer dentro do intervalo interquatílico. Esse procedimento também será realizado no dataframe “df_ln”.

Também será feita a cópia desses dataframes (“df_out” e “df_ln_out”) a fim de comparação final dos resultados de aplicação dos modelos comparando dataframes com outliers e sem outiliers. Abaixo segue a comparação da distribuição dos outliers e da distribuição das novas variáveis “Preco_Area” e “Preco_Area_ln”.

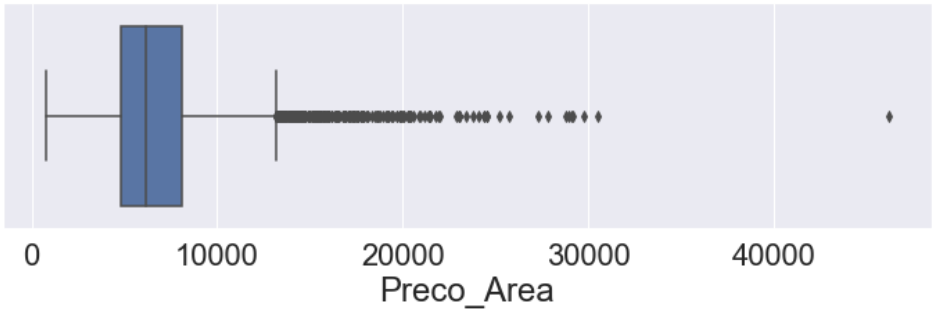
```
# A variável Area é a que mais influencia no Preco.
df_out = df.copy()
df_ln_out = df_ln.copy()

df['Preco_Area'] = round(df['Preco'] / df['Area'],2)
df_ln['Preco_Area_ln'] = round(df_ln['Preco_ln'] / df_ln['Area_ln'],2)

sns.displot(data=df, x='Preco_Area', kde=True,height=5),
sns.displot(data=df_ln, x='Preco_Area_ln', kde=True,height=5)
```



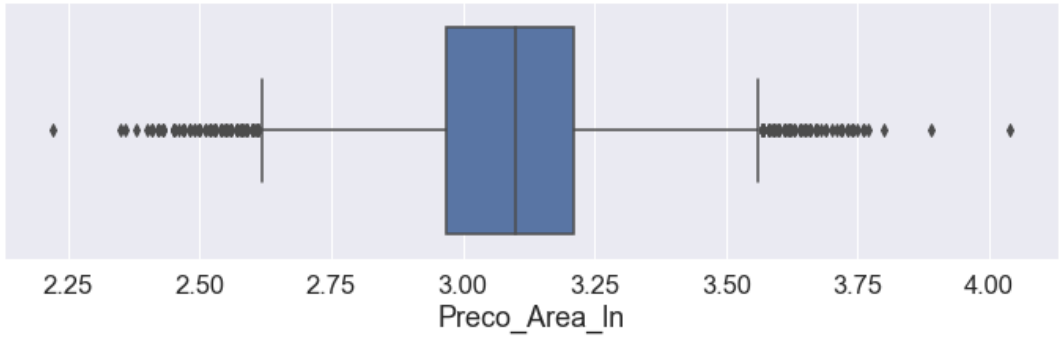


Através da função .boxplot() visualiza-se a diferença entre as duas variáveis na distribuição de possíveis outliers.

```
AxesSubplot(0.125,0.125;0.775x0.755)
```




```
AxesSubplot(0.125,0.125;0.775x0.755)
```



Aplica-se a função *outlier_iqr(df,cols)* que retornará os limites mínimo e máximo dos outliers, e informará a quantidade e o percentual de registros que são outliers da coluna referente ao preço por metro quadrado (“Preco_Area”).

```
# Aplicação da função outlier_iqr()
cols = ['Preco_Area']
low_out,upp_out = outlier_iqr(df, cols)

cols = ['Preco_Area_ln']
low_out_ln,upp_out_ln = outlier_iqr(df_ln, cols)
```

```
Outliers em Preco_Area 4.57% Total de registros: 288
Outliers em Preco_Area_ln 3.33% Total de registros: 210
```

Foram calculados 288 *outliers* no “df” e 210 *outliers* no “df_ln”. A seguir será aplicado método para excluir esses registros e posteriormente será usada a função .drop() para excluir a coluna “Preco_Area” e “Preco_Area_ln” dos respectivos dataframes.

```
# Cálculo do Intervalo Interquatílico (IQR) na variável Preco_Area nos datasets df e df_ln:

df = df[df['Preco_Area'] > low_out]
df = df[df['Preco_Area'] < upp_out]

df_ln = df_ln[df_ln['Preco_Area_ln'] > low_out_ln]
df_ln = df_ln[df_ln['Preco_Area_ln'] < upp_out_ln]

df.drop(columns = ['Preco_Area'], inplace = True)
df_ln.drop(columns = ['Preco_Area_ln'], inplace = True)

print('Shape do df:', df.shape)
print('Shape do df_ln:', df_ln.shape)
```

```
Shape do df: (6013, 11)
Shape do df_ln: (6091, 11)
```

4.2.7 Padronização de dados

Finalizando essa etapa, segue a padronização do conjunto de dados a ser aplicado no dataframe “df”. A padronização é um método útil que dimensiona variáveis independentes para que tenham uma distribuição com valor médio 0 e variância igual a 1. Dentre os diversos métodos para padronização, os mais comuns são *o StandarScaler(), MinMaxScaler()* e o *RobustScaler()*. Como há dados que não são normalmente distribuídos, o *StandardScale*r não é uma boa opção para esse caso. Já a normalização *MinMaxScaler*, todos os valores são normalizados no intervalo entre 0 e 1. Este método, no entanto, tem a desvantagem de ser muito sensível aos outliers. O RobustScaler usa estimativas mais robustas para o centro e o alcance dos dados, pois remove a mediana e dimensiona os dados de acordo com o intervalo interquartil (IQR). O IQR é o intervalo entre o 1º quartil (25º quantil) e o 3º quartil (75º quantil). Por essas características, optou-se pelo *RobustScaler* para realizar a padronização dos dados.

```
# Aplicando RobustScaler em df e df_out
cols = list(df.columns)
df_r = RobustScaler().fit_transform(df)
df_r = pd.DataFrame(df_r, columns=cols)
df= df_r.copy()

cols = list(df_out.columns)
df_r = RobustScaler().fit_transform(df_out)
df_r = pd.DataFrame(df_r, columns=l_select_cols)
df_out = df_r.copy()

df_out.head()
```

	Preco	Area	Garagem	Condo	Banheiros	Mais de 20_SM_%	Quartos	Longitude	População (2010)	Piscina
0	0.82	0.32	1.00	1.56	0.00	1.73	-1.00	-0.59	-0.46	0.00
1	3.74	2.68	2.00	5.94	3.00	1.73	2.00	-0.65	-0.46	0.00
2	0.79	0.21	0.00	0.62	0.00	1.73	0.00	-0.59	-0.46	0.00
3	3.02	2.45	1.00	3.38	3.00	1.73	1.00	-0.58	-0.46	0.00
4	1.90	-0.16	1.00	1.25	0.00	1.73	0.00	-0.64	-0.46	0.00

Finalizando essa etapa, é feita exclusão de dados duplicados e a verificação se há dados nulos e qual a dimensão dos respectivos dataframes que alimentarão os modelos de aprendizado de máquina. Importante que todos os dataframes tenham o mesmo número de colunas e nenhum dado nulo.

```
# Excluindo dados duplicados

df_o = df.drop_duplicates()
df_ln = df_ln.drop_duplicates()
df_out = df_out.drop_duplicates()
df_ln_out = df_ln_out.drop_duplicates()

#Verificando se há dados nulos
print('Totais de dados nulos\ndf: %s \ndf_ln: %s \ndf_out: %s  \ndf_ln_out: %s'
      '\n\nDimensão dos DataFrames\ndf: %s \ndf_ln: %s \ndf_out: %s \ndl_ln_out:%s'
      %(df_o.isna().sum().sum(), df_ln.isna().sum().sum(), df_out.isna().sum().sum(),
        df_ln_out.isna().sum().sum(), df_o.shape,df_ln.shape, df_out.shape, df_ln_out.shape))
```

```
Totais de dados nulos
df: 0
df_ln: 0
df_out: 0
df_ln_out: 0

Dimensão dos DataFrames
df: (5988, 11)
df_ln: (6063, 11)
df_out: (6276, 11)
dl_ln_out:(6273, 11)
```

5. Criação de Modelos de Machine Learning

Neste tópico serão apresentados os modelos de aprendizado supervisionado utilizando os recursos da biblioteca ScikitLearn. Como o objetivo é predizer o preço de venda de apartamentos foram escolhidos os seguintes modelos:

- Regressão linear: Mínimos Quadrados Ordinários, Ridge, Lasso e Elastic-Net;

- Regressão baseada em k vizinhos mais próximos (KNeighborsRegressor);

- Árvores de decisão (DecisionTreeRegressor);

- Métodos de conjunto (ensemble methods): RandomForestRegressor, GradientBoostingRegressor e AdaBoostRegressor.

Como o desempenho de um modelo depende significativamente do valor dos hiperparâmetros, será usado *RandomizedSearchCV()* para cada um dos modelos, a fim de ajustar os hiperparâmetros. O que permitirá determinar os valores ideais a serem usados em cada dataset(“df”, “df_ln”, “df_out” e “df_ln_out”). O *RandomizedSearchCV()* é uma função que vem no pacote *model_selection* do Scikit-learn, ele testa diversos valores aleatoriamente permitindo ao final selecionar os melhores parâmetros dos hiperparâmetros previamente listados.

Esse processo de ajuste dos hiperparâmetros será otimizado por validação cruzada (*Cross Validation-CV)*, que consiste em dividir os dados em *n* conjuntos, onde *n-1* conjuntos são usados para treino e o conjunto que sobra é usado para teste e avaliação do desempenho do modelo. Esse procedimento é realizado n vezes para cada um dos conjuntos (todos os conjuntos serão usados para teste uma vez), gerando um intervalo de métricas (*scores)*.

5.1 *Overfitting* e *Underfitting*

Um dos principais problemas na construção de modelos de predição é evitar a ocorrência de "*overfitting*" (ajuste excessivo) ou de "*underfitting*" (ajuste insuficiente).

Underfitting (subajuste) significa que o modelo não conseguiu aprender o suficiente sobre os dados. O underfitting leva à um erro elevado em ambos os dados de treino e de teste. Esse problema é mais fácil de ser identificado.

Overfitting (superajuste) é o oposto do *underfitting*. É quando o modelo aprende demais sobre os dados. Neste caso, o modelo mostra-se adequado apenas para os dados de treino, não sendo capaz de generalizar para os dados de teste. O *overfitting* pode ser identificado quando há grande diferença de performance do modelo entre os dados de treino e de teste, sendo mais difícil de se identificar.

O objetivo dos modelos de Machine Learning é estimar a função que melhor ajusta aos dados de entrada para obter previsões corretas de forma generalizada. A melhor maneira de medir e otimizar o desempenho do modelo é o de balancear a relação entre o viés (*bias*) e variância(*variance*).

Bias é o erro devido à diferença entre as previsões médias e os valores corretos. O Bias é a distância, em geral, das previsões para o valor correto. Bias elevado indica que o modelo se ajusta pouco aos dados de treino, causando *underfitting*. O que significa que o Erro Quadrado Médio (MSE – *Mean Squared Error)* é alto para a base de teste.

Variance é o erro devido à variabilidade de uma previsão do modelo para um determinado ponto de dados. A variância mede o quanto as previsões do modelo variam entre diferentes realizações do modelo. *Variance* alta diz que o modelo se ajusta demais aos dados

de treino, causando *overfitting*. Nesse caso o MSE é zero para os dados de treino e não generaliza bem os dados.

5.2 Métricas de desempenho

Bons modelos de predição precisam apresentar uma boa relação entre os erros bias e *variance,* o modelo ideal seria aquele que tem baixo viés e baixa variância. Para avaliar o desempenho dos modelos, as métricas mais comuns são o erro quadrado médio, erro médio absoluto, erro quadrático médio e R-quadrado ou coeficiente de determinação.

O erro absoluto médio é a média da diferença absoluta entre os valores reais e previstos no conjunto de dados. O algoritmo soma todas as diferenças (em valores absolutos) em todos os preços previstos e reais e depois divide pelo número de observações. É a média dos resíduos no conjunto de dados, sendo que um valor mais baixo indica melhor precisão.

O Erro Quadrado Médio representa a média da diferença quadrada entre os valores original e previsto no conjunto de dados. Ele mede a variância dos resíduos. Sendo o erro quadrático médio a raiz quadrada do erro quadrado médio, sendo a medida do desvio padrão dos resíduos. Tanto o erro médio quadrado (MSE) como o erro quadrático médio (RMSE) penalizam os grandes erros de previsão em relação ao erro médio absoluto (MAE), pois como os erros são elevados ao quadrado antes de ter a média calculada, conforme os valores de erros aumentam, Tanto MSE e RMSE aumentaram consideravelmente.

O coeficiente de determinação ou R-quadrado é a proporção da variância na variável dependente que é explicada pelo modelo, informando o quanto as variáveis preditoras explicam a variação na variável resposta. De forma mais simples, o R-quadrado é a porcentagem da variação da variável resposta que é explicada pelo modelo. O valor de R quadrado sempre estará entre 0 e 1 (0% e 100%).

Assim, menores valores de MAE, MSE e RMSE são desejáveis pois implicam em maior precisão de um modelo. Já em relação ao R quadrado, busca-se valores mais altos. Dessa forma, as métricas de erro médio quadrático, erro médio absoluto e R-quadrado são usadas para avaliar o desempenho dos modelos escolhidos.

5.3 Descrição da estrutura básica das funções de *Machine Learning*

No desenvolvimento deste trabalho foi desenvolvida uma função específica para cada algoritmo de aprendizado de máquina. Essa função calculará os melhores hiperparâmetros para cada dataset, na sequência irá estimar a acurácia com validação cruzada e, tendo já calculado os melhores hiperparâmetros, treinará o modelo com os dados de treino e teste retornando todas as métricas necessárias para avaliar a acurácia do modelo. No final será retornado um dataframe com todos os modelos testados e suas respectivas métricas.

Segue descrição da estrutura básica para a chamada das funções criadas para os modelos de *machine learning*.

modelos, scores_map, resultados = função(dataset,dataframe,modelos, scores_map, kf, rs)

modelos, scores_map e resultados são as variáveis a serem retornadas pela função:

- modelos: dicionário contendo os modelos com os melhores parâmetros estimados;

- *scores_map*: dicionário contendo o intervalo das acurácias gerado pelos resultados obtidos com a função de validação cruzada (cross_val_score()), aplicado com os melhores parâmetros calculados;

- resultados : um dicionário informando as métricas de treino e de teste (MAE,RMSE, R^2), modelo e o nome do *dataset* usado.

Os argumentos necessários para ativar a função são:

-*dataset*: o nome do dataframe, variável tipo string(str);

-dataframe: o próprio dataframe;

-*ml_functions*: lista contendo o nome de cada função criada para o processamento dos *datasets*;

-modelos: dicionário contendo os modelos de *machine learning* com os melhores parâmetros já testados para cada dataframe;

-*scores_map*: dicionário contendo o intervalo de valores de MSE calculado gerado pelos resultados obtidos com cross_val_score(), aplicado com os melhores parâmetros calculados;

-*kf*: número de divisões dado pela função *KFold().* Número de fatias escolhido foi 5 e *True* para embaralhamento(*shuffle*);

-*rs*: número aleatório (*random state*) para padronizar os resultados. Nos testes foram usados vários números, para padronizar essa apresentação foi escolhido o número 2000(dois mil).

```
modelos={'df':[],'df_ln':[],'df_out':[],'df_ln_out':[]}
scores_map = {'df':{},'df_ln':{},'df_out':{},'df_ln_out':{}}
kf = KFold(n_splits=5, shuffle=True)
rs = 2000

ml_functions = ['K_Neighbors_Regressor', 'Random_Forest_Regressor','Decision_Tree_Regressor', 'Reg_Linear',
                'Ridge_', 'Lasso_','Elastic_Net', 'AdaBoost_Regressor', 'Gradient_Boosting_Regressor']

datasets = {'df':df, 'df_ln':df_ln, 'df_out':df_out, 'df_ln_out':df_ln_out}


df_compara = pd.DataFrame(columns = ['Dataframe', 'Modelo', 'MAE - Treina', 'MAE - Teste', 'RMSE - Treina',
                                     'RMSE - Teste', 'R2 (Acurácia) - Treina', 'R2 (Acurácia) - Teste'])
```

O dataframe “df_compara” é criado vazio e receberá todos os resultados de todos os modelos testados nos quatro datasets (df, df_ln, df_out e df_ln_out) permitindo visualizar em qual dataset e com qual modelo obtêm-se a melhor acurácia.

Por dentro de cada função de *machine learning,* cria-se as variáveis x e y, contendo as variáveis preditoras e resposta (preço) respectivamente. Segue-se a aplicação da função *train_test_split()* adotando a divisão 80/20 entre dados de treino e dados de teste. A seguir é criado o dicionário de parâmetros (*parameters*) informando os mais diversos parâmetros que alimentaram a função *RandomizedSearchCV().* Esses parâmetros irão variar conforme o modelo a ser treinado. A seguir a variável model que conterá a função do modelo que será treinado.

A variável “rand_cv” contém a função *RandomizedSearchCV()* que foi escolhido para descobrir qual conjunto de hiperparâmetros entrega o melhor resultados. Essa função é alimentada pela variável model, cv (recebe a variável kf) *param_distributions* (recebe *parameters*). Como *scoring* foi escolhido o MSE negativo (‘*neg_mean_squared_error*’), o qual segue a convenção de que valores de retorno mais altos são melhores que valores mais baixos. Assim, um grande MSE mais alto é melhor do que um baixo.

A seguir aplica-se a função *.fit()* com os dados de treino na variável rand_cv. Abaixo segue imagem da função preparada para treinar regressão linear.

```
x = df.drop(columns='Preco')
y = df['Preco']
train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2)
parameters = {'n_jobs':[-1,1,2,3,5,10,25], "fit_intercept": [True, False],'normalize':[False, True]}

# define modelo/ estimador
model = LinearRegression()

# definindo Rand search
rand_cv= RandomizedSearchCV(model, param_distributions=parameters, scoring='neg_mean_squared_error',cv=kf)

#fit no Rand search
rand_cv.fit(train_x,train_y)
```

A função *cross_val_score()* realizará uma validação cruzada nos dados de treino. Terá como argumentos a variável model, contendo o modelo com os melhores hiperparâmetros estimados, os dados de treino (“train_x” e “train_y”) o número de divisões(“kf”) e o *scoring* (‘*neg_mean_squared_error*’). O resultado será atribuído a variável scores, cujo valor será adicionado ao dicionário “scores_map”. Aplica-se print informando os melhores hiperparâmetros estimados com o método *.best_estimator_*.

```
scores = cross_val_score(model, train_x, train_y, cv=kf, scoring='neg_mean_squared_error')
d_scores={}
d_scores['Ridge'] = scores
scores_map[dataset].update(d_scores)

# melhor estimador
print("Melhor classificação :", rand_cv.best_estimator_)
```

A seguir é realizado o modelo é treinado com os melhores parâmetros. São salvos os valores de RMSE, MAE e R^2 de treino e na sequência o modelo é aplicado para os dados de teste, salvando os valores de teste RMSE, MAE e R^2 respectivamente.

```
# melhor modelo
modelo = rand_cv.best_estimator_
modelo.fit(train_x,train_y)

y_pred = modelo.predict(train_x)
# Calcula a métrica Root Mean Squared Error
rmse_train=sqrt(mean_squared_error(train_y,y_pred))
#Calcula a métrica Mean Absolute Error
mae_train = mean_absolute_error(train_y,y_pred)
score_train = modelo.score(train_x, train_y)*100

modelo.fit(train_x,train_y)
y_pred = modelo.predict(test_x)
rmse_test=sqrt(mean_squared_error(test_y,y_pred))
mae_test = mean_absolute_error(test_y,y_pred)
score_test = modelo.score(test_x, test_y)*100
MAPE = np.mean(np.abs((test_y - y_pred) / abs(test_y))) * 100
```

O modelo é incluído na lista modelos e são adicionados no dicionário resultados os seguintes registros:

Dataframe: nome do dataframe calculado;

Modelo: recebe a variável ‘modelo’

MAE – Treina: recebe a variável ‘mae_train’

MAE – Teste: recebe a variável ‘mae_test’;

RMSE – Treina: recebe a variável ‘rmse_train’

RMSE – Teste: recebe a variável ‘rmse_test’

R2 (Acurácia) – Treina: recebe a variável ‘score_train’

R2 (Acurácia) – Teste: recebe a variável ‘score_test’

Esse dicionário irá fornecer os dados para montar dataframe(“df_compara”) para comparar os resultados finais.

```
    modelos[dataset].append(modelo)

    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test], 'R2 (Acurácia) - Treina':[score_train],
                  'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

O “scores_map” permite visualizar qual modelo tem menor variância nos resultados plotando os scores obtidos pela aplicação da função *cross_val_score()*, indicando quais modelos seriam mais robustos.

Abaixo segue a imagem da célula com as definições das variáveis que alimentam as funções dos modelos já definidos. Também há os comandos *"for in"* necessários para ativar as funções em ordem preestabelecida.

```
modelos={'df':[],'df_ln':[],'df_out':[],'df_ln_out':[]}
scores_map = {'df':{},'df_ln':{},'df_out':{},'df_ln_out':{}}
kf = KFold(n_splits=5, shuffle=True)
rs = 2000

ml_functions = ['K_Neighbors_Regressor', 'Random_Forest_Regressor','Decision_Tree_Regressor', 'Reg_Linear',
                'Ridge_', 'Lasso_','Elastic_Net', 'AdaBoost_Regressor', 'Gradient_Boosting_Regressor']

datasets = {'df':df, 'df_ln':df_ln, 'df_out':df_out, 'df_ln_out':df_ln_out}


df_compara = pd.DataFrame(columns = ['Dataframe', 'Modelo', 'MAE - Treina', 'MAE - Teste', 'RMSE - Treina',
                                     'RMSE - Teste', 'R2 (Acurácia) - Treina', 'R2 (Acurácia) - Teste'])

for function in ml_functions:
    for dataset,dataframe in datasets.items():
            modelos, scores_map,resultados = globals()[function](dataset,dataframe,modelos, scores_map, kf,rs)
            df_mod=pd.DataFrame(data = resultados)
            df_compara=pd.concat([df_compara,df_mod])
```

5.4 Funções de *Machine Learning*

5.4.1 Regressão Linear

A Regressão Linear dos Mínimos Quadrados Ordinários é um modelo simples de *machine learning* para encontrar a reta que melhor explica a relação entre as variáveis preditoras e uma variável de resposta. A função *LinearRegression*() do *ScikitLearn* ajusta um modelo linear com coeficientes w = (w1, …, wp) para minimizar a soma residual dos quadrados entre os alvos observados no conjunto de dados e os alvos previstos pela aproximação linear, retornando o coeficiente de determinação da previsão.

Os parâmetros usados para fazer o teste no *RandomizedSearchCV* foram:

n_jobs: parâmetro é usado para especificar quantos processos ou encadeamentos simultâneos devem ser usados para rotinas paralelizadas

fit_intercept: calcular o intercepto para este modelo. Se definido como False, nenhum intercepto será usado nos cálculos(ou seja, espera-se que os dados sejam centrados).

normalize: Este parâmetro é ignorado quando *fit_intercept* é *False*. Se *True*, os regressores X serão normalizados antes da regressão.

Abaixo segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def Reg_Linear(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)
    parameters = {'n_jobs':[-1,1,2,3,5,10,25], "fit_intercept": [True, False],'normalize':[False, True]}
    # define modelo/ estimador
    model = LinearRegression()
    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters, scoring='neg_mean_squared_error',
                                cv=kf, random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['Regressão_Linear'] = scores
    scores_map[dataset].update(d_scores)

    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.2 *Ridge e Lasso*

Dentro da regressão linear, Ridge e Lasso são formas de regularizar a função através de penalidades. De forma simples, dentro de uma equação estatística dos dados, alteram-se os fatores de forma a priorizar ou não certas parcelas da equação e, assim, melhorar a qualidade de predição. A Regularização remove os ruídos e deixa o modelo mais simples, de melhor performance, generalizável e com boa acurácia.

A regularização L1 (Lasso - operador de seleção e encolhimento mínimo absoluto) é um método de análise de regressão que realiza seleção e regularização de variáveis para melhorar a precisão do modelo, ou seja, quando há múltiplas variáveis altamente correlacionadas, apenas uma delas é selecionada os coeficientes das outras são zerados, minimizando a penalização L1. Desse modo, esse modelo realiza *feature selection* automaticamente, reduzindo o ruído do modelo.

A regularização L2 (*Ridge*) resulta em valores de peso menores, o que estabiliza os pesos quando há alta correlação entre as variáveis. a penalização L2 é desproporcionalmente maior para coeficientes maiores, a regularização Ridge faz com que as variáveis correlacionadas tenham coeficientes parecidos. Dessa forma, Ridge não diminui o impacto dos outliers, sendo mais apropriado remover as variáveis desnecessárias antes de aplicar essa regressão.

Os parâmetros usados para fazer o teste no *RandomizedSearchCV* tanto para Lasso e Ridge foram:

Alpha (Força de regularização): deve ser um *float* positivo. A regularização melhora o condicionamento do problema e reduz a variância das estimativas.

fit_intercept: calcular o intercepto para este modelo. Se definido como False, nenhum intercepto será usado nos cálculos(ou seja, espera-se que os dados sejam centrados).

normalize: Este parâmetro é ignorado quando *fit_intercept* é *False*. Se *True*, os regressores X serão normalizados antes da regressão.

Max_iter: Número máximo de iterações para o solucionador de gradiente conjugado.

Abaixo segue imagem das funções Ridge e Lasso, que retornam o resultado para os melhores parâmetros calculados em cada dataframe:

```
# Função Ridge
def Ridge_(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)
    parameters = {'alpha':[0.0001,0.001,0.01,0.05,0.1,1, 10,100],'max_iter':[10,20,50,100,200,300,500,700,1000,2000,
                  'normalize':[False, True], 'fit_intercept':[False, True]}
    # define modelo/ estimador
    model = Ridge()

    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters,scoring='neg_mean_squared_error',
                                cv=kf, random_state=rs)

    #fit no Rand search
    rand_cv.fit(train_x,train_y)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['Ridge'] = scores
    scores_map[dataset].update(d_scores)
    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

```
def Lasso_(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)
    parameters = {'alpha':[0.0001,0.001,0.01,0.05,0.1,1, 10,100],'max_iter':[10,20,50,100,200,300,500,700,1000,2000,5000,10000],
                  'normalize':[False, True], 'fit_intercept':[False, True]}

    # define modelo/ estimador
    model = Lasso()
    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters,scoring='neg_mean_squared_error',
                                cv=kf, random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores = {}
    scores_map['Lasso'] = scores
    scores_map[dataset].update(d_scores)
    print(scores)
    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)

    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.3 *ElasticNet* – L1+L2

ElasticNet combina os termos de regularização de L1 e L2. Essa combinação permite aprender um modelo esparso onde poucos dos pesos são diferentes de zero como Lasso, mantendo as propriedades de regularização de Ridge. O *ElasticNet* é útil quando há vários recursos correlacionados entre si. Os parâmetros usados para fazer o teste no *RandomizedSearchCV* são os mesmos de *Ridge* e *Lasso*, adicionando-se *l1_ratio*, que é um parâmetro de penalidades entre L1 e L2, variando de 0 a 1. Os parâmetros dessa função são:

Alpha (Força de regularização): deve ser um *float* positivo. A regularização melhora o condicionamento do problema e reduz a variância das estimativas.

fit_intercept: calcular o intercepto para este modelo. Se definido como False, nenhum intercepto será usado nos cálculos(ou seja, espera-se que os dados sejam centrados).

normalize: Este parâmetro é ignorado quando fit_intercept é False. Se True, os regressores X serão normalizados antes da regressão.

Max_iter: Número máximo de iterações para o solucionador de gradiente conjugado.

l1_ratio: é o parâmetro de mixagem *ElasticNet*. Para 0 < *l1_ratio* < 1, a penalidade é uma combinação de L1 e L2. Para *l1_ratio* = 0, é uma penalidade L2 e para *l1_ratio* = 1 é a penalidade L1.

Abaixo segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def Elastic_Net(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)
    parameters = {'alpha':[0.0001,0.001,0.01,0.05,0.1,1, 10,100],'max_iter':[10,20,50,100,200,300,500,700,1000,2000,5000,10
                  'l1_ratio': np.arange(0.0, 1.0, 0.1),'fit_intercept':[False, True],'normalize':[False, True]}
    # define modelo/ estimador
    model = ElasticNet()

    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions= parameters,cv=kf, random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)

    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['ElasticNet'] = scores
    scores_map[dataset].update(d_scores)

    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.4 *Decision Tree Regression*

As Árvores de Decisão (*DecisionTree*) são um método de aprendizado supervisionado não paramétrico e são divididas em árvores de classificação e de regressão. Árvores de classificação são necessárias quando a variável resposta é um conjunto discreto de valores, já as árvores de regressão são usadas quando a variável resposta são valores numéricos ou contínuos. O objetivo é criar um modelo que preveja o valor de uma variável de destino aprendendo regras de decisão simples inferidas a partir dos recursos de dados. Uma árvore pode ser vista como uma aproximação constante por partes.

No processo de aprendizagem da árvore de decisão podem ser criadas árvores supercomplexas que não generalizam bem os dados (*overfitting*). Para evitar esse problema podem ser adotados mecanismos como poda, definição do número mínimo de amostras necessárias em um nó folha ou definir apropriadamente a profundidade máxima da árvore (parâmetro *max_depth*). Os parâmetros usados para fazer o teste no *RandomizedSearchCV* são:

Max_depth: A profundidade máxima da árvore. Se for muito alta, as árvores de decisão aprendem detalhes muito finos dos dados de treinamento, havendo o superajuste(overfitting);

Min_samples_leaf: O número mínimo de amostras necessárias para estar em uma folha, garantindo que cada folha tenha um tamanho mínimo;

Min_samples_split: O número mínimo de amostras necessárias para dividir um nó interno;

Segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def Decision_Tree_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)
    parameters = {'max_depth':[3,5,10,15,20,30,None],'min_samples_leaf':randint(32, 128),
                  "min_samples_split":randint(32,128)}

    model = DecisionTreeRegressor()
    rand_cv = RandomizedSearchCV(model, cv=kf, param_distributions=parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)

    #fit no Rand search
    rand_cv.fit(train_x, train_y)
    print("Melhor classificação :", rand_cv.best_estimator_)
    scores = cross_val_score(model,x, y, cv=kf)
    d_scores={}
    d_scores['DecisionTreeReg'] = scores
    scores_map[dataset].update(d_scores)
    # melhor modelo

    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100

    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.5 *Random Forest Regressor*

A Floresta Aleatória (*random forest*) é um algoritmo de aprendizagem supervisionada não paramétrico e pode ser usado para problemas de classificação e de regressão. A “floresta” que ele cria é uma combinação (ensemble) de árvores de decisão de um modo aleatório. Essa característica torna esse algoritmo mais robustos e complexos, levando a um maior custo computacional. Sendo essa a principal diferença com o modelo *Decision Tree*. Assim, o modelo Randon Forest trabalha com subconjuntos aleatórios e constrói árvores menores a partir de tais subconjuntos, sendo essas árvores combinadas depois do treinamento, por consequência, diminuindo drasticamente as chances de *overfitting*. Os parâmetros usados para fazer o teste no *RandomizedSearchCV* são:

n_estimators: O número de árvores a serem construídas pelo algoritmo.

Max_depth: A profundidade máxima da árvore. Se Nenhum, os nós são expandidos até que todas as folhas sejam puras ou até que todas as folhas contenham menos de *min_samples_split* samples.

Min_samples_leaf: número mínimo de amostras para estar em dada árvore.

Min_samples_split: número mínimo de amostras para dividir um nó interno.

Abaixo segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def Random_Forest_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)

    parameters={'n_estimators':[50,100,150,200], 'max_depth':[1,2,3,5,6,7,8,9,10,15,20,None],
                'min_samples_leaf':randint(32,128), "min_samples_split":randint(32,128)}
    # define modelo/ estimador
    model = RandomForestRegressor()
    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters,
                                scoring='neg_mean_squared_error',cv=kf, random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)

    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['RandomForestReg'] = scores
    scores_map[dataset].update(d_scores)

    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.6 K-*Nearest Neighbors*

Em estatística, o algoritmo de k-vizinhos mais próximos (k-NN) é um método de aprendizado supervisionado não paramétrico usado para classificação e regressão. Em ambos os casos, a entrada consiste nos k exemplos de treinamento mais próximos em um conjunto de dados. A saída depende se k-NN é usado para classificação ou regressão. Na regressão k-NN, a saída é o valor da propriedade para o objeto. Este valor é a média dos valores de k vizinhos mais próximos. k-NN é um tipo de classificação onde a função é apenas aproximada localmente e toda a computação é adiada até a avaliação da função. Tanto para classificação quanto para regressão, uma técnica útil pode ser atribuir pesos às contribuições dos vizinhos, de modo que os vizinhos mais próximos contribuam mais para a média do que os mais distantes. Por exemplo, um esquema de ponderação comum consiste em dar a cada vizinho um peso de 1/d, onde d é a distância ao vizinho. Os parâmetros usados para fazer o teste no *RandomizedSearchCV* são:

n_neighbors: Número de vizinhos a serem usados por padrão para consultas.

weights: Função de peso usada na previsão.

Algorithm: Algoritmo usado para calcular os vizinhos mais próximos.

p: Parâmetro de potência para a métrica *Minkowski*. Quando p = 1, isso é equivalente a usar distância_*manhattan* (L1) e distância_euclidiana (L2) para p = 2.

Abaixo segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def K_Neighbors_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)
    parameters = {"n_neighbors" : [2,3,4,5,6,7,8,9,10,12,15],'weights':['uniform', 'distance'],
                  'algorithm':['auto', 'ball_tree', 'kd_tree', 'brute'], 'p':[1,2]}

    model = KNeighborsRegressor()
    rand_cv = RandomizedSearchCV(model, cv=kf, param_distributions=parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x, train_y)

    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['KNeighborsReg'] = scores
    scores_map[dataset].update(d_scores)

    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.7 ***AdaBoostingRegressor***

AdaBoost, abreviação de "*Adaptive Boosting*", é um algoritmo de aprendizado de máquina de conjunto de reforço e foi uma das primeiras abordagens de reforço bem-sucedidas.

Boosting é uma classe de algoritmos de aprendizado de máquina que envolvem a combinação das previsões de muitos modelos preditivos fracos. Um modelo fraco é um modelo muito simples, embora tenha alguma habilidade no conjunto de dados. *Boosting* era um conceito teórico muito antes de um algoritmo prático ter sido desenvolvido, e o algoritmo *AdaBoost* (aumento adaptativo) foi a primeira abordagem bem-sucedida para a ideia.

Esse algoritmo envolve o uso de árvores de decisão muito curtas (de um nível) como aprendizes fracos que são adicionados sequencialmente ao conjunto. Cada modelo subsequente tenta corrigir as previsões feitas pelo modelo anterior na sequência. Isso é obtido pesando o conjunto de dados de treinamento para colocar mais foco nos exemplos de treinamento nos quais os modelos anteriores cometeram erros de previsão.

O algoritmo foi desenvolvido para classificação e envolve a combinação das previsões feitas por todas as árvores de decisão do ensemble. Uma abordagem semelhante também foi desenvolvida para problemas de regressão onde as previsões são feitas usando a média das árvores de decisão. A contribuição de cada modelo para a previsão do conjunto é ponderada com base no desempenho do modelo no conjunto de dados de treinamento. Os parâmetros usados para fazer o teste no *RandomizedSearchCV* são esses:

n_estimators: O número máximo de estimadores em que o reforço é encerrado. Em caso de ajuste perfeito, o processo de aprendizagem é interrompido precocemente.

learning_rate: Peso aplicado a cada regressor em cada iteração de reforço. Uma taxa de aprendizado mais alta aumenta a contribuição de cada regressor. Há um *trade-off* entre os parâmetros learning_rate e n_estimators.

loss: A função de perda a ser usada ao atualizar os pesos após cada iteração de reforço.

Segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def AdaBoost_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    dataframe.dropna(inplace=True)
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)

    model = AdaBoostRegressor()
    parameters={'n_estimators':[50,100, 150,200,250],
                'learning_rate':[0.001,0.01,0.05,0.1,1,5,10,100],
                'loss':['linear', 'square', 'exponential']}
    rand_cv = RandomizedSearchCV(model, cv=kf,param_distributions= parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x, train_y)
    print("Melhor classificação:", rand_cv.best_estimator_)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['AdaBoostReg'] = scores
    scores_map[dataset].update(d_scores)

    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100

    modelos[dataset].append(modelo)

    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test], 'R2 (Acurácia) - Treina':[score_train],
                  'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

5.4.8 *Gradient Boosting Regressor*

O algoritmo de aumento de gradiente é usado para gerar um modelo de conjunto combinando os modelos preditivos fracos. O aumento de gradiente cria um modo aditivo usando várias árvores de decisão de tamanho fixo como modelos preditivos fracos. O parâmetro, *n_estimators*, decide o número de árvores de decisão que serão utilizadas nos estágios de *boosting*. A principal diferença entre *Gradient Boosting* e o *AdaBoost* é que o primeiro é mais robusto (menos sensível) para valores discrepantes. O processo de ajuste do modelo começa com a constante como o valor médio dos valores alvo. Nas etapas subsequentes, as árvores de decisão ou os estimadores são ajustados para prever os gradientes negati-

vos das amostras. Os gradientes são atualizados em cada iterador (para todos os estimadores subsequentes). Uma taxa de aprendizado é usada para reduzir o resultado ou a contribuição de cada árvore ou estimador subsequente. Os parâmetros usados para fazer o teste no *RandomizedSearchCV* são:

n_estimators: O número de estágios de reforço a serem executados. O aumento de gradiente é bastante robusto ao ajuste excessivo, portanto, um número grande geralmente resulta em melhor desempenho

learning_rate: A taxa de aprendizado reduz a contribuição de cada árvore por *learning_rate*. Há um *trade-off* entre *learning_rate* e *n_estimators*

min_samples_leaf: O número mínimo de amostras necessárias para estar em um nó folha. Um ponto de divisão em qualquer profundidade só será considerado se deixar pelo menos amostras de treinamento *min_samples_leaf* em cada um dos ramos esquerdo e direito. Isso pode ter o efeito de suavizar o modelo, especialmente na regressão.

min_samples_split: O número mínimo de amostras necessárias para dividir um nó interno

max_depth: Profundidade máxima dos estimadores de regressão individuais. A profundidade máxima limita o número de nós na árvore. Ajuste este parâmetro para melhor desempenho; o melhor valor depende da interação das variáveis de entrada.

Segue imagem da função que retorna os melhores resultados para os melhores parâmetros calculados em cada dataframe:

```
def Gradient_Boosting_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    dataframe.dropna(inplace=True)
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_state=rs)

    model = GradientBoostingRegressor()
    parameters={'n_estimators':[50,100, 150,200,250],
                'learning_rate':[0.001,0.01,0.05,0.1,1,5,10],
                "min_samples_split" : randint(32, 128),  "max_depth" : [3, 5, 10, 15, 20, 30, None],
                "min_samples_leaf" : randint(32, 128)}

    rand_cv = RandomizedSearchCV(model, cv=kf, param_distributions=parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x, train_y)
    print("Melhor classificação:", rand_cv.best_estimator_)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['GradientBoostingReg'] = scores
    scores_map[dataset].update(d_scores)

    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100

    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

6. Interpretação dos Resultados

Aplicando as funções ora definidas pra cada dataframe, foi gerado o dataframe 'df_compara' que relaciona todas as métricas nas bases de treino e teste (RMSE, MAE e R^2), respectivos datafra-mes e modelos com seus respectivos parâmetros. Abaixo segue imagem com o esquema para gerar 'df_compara'.

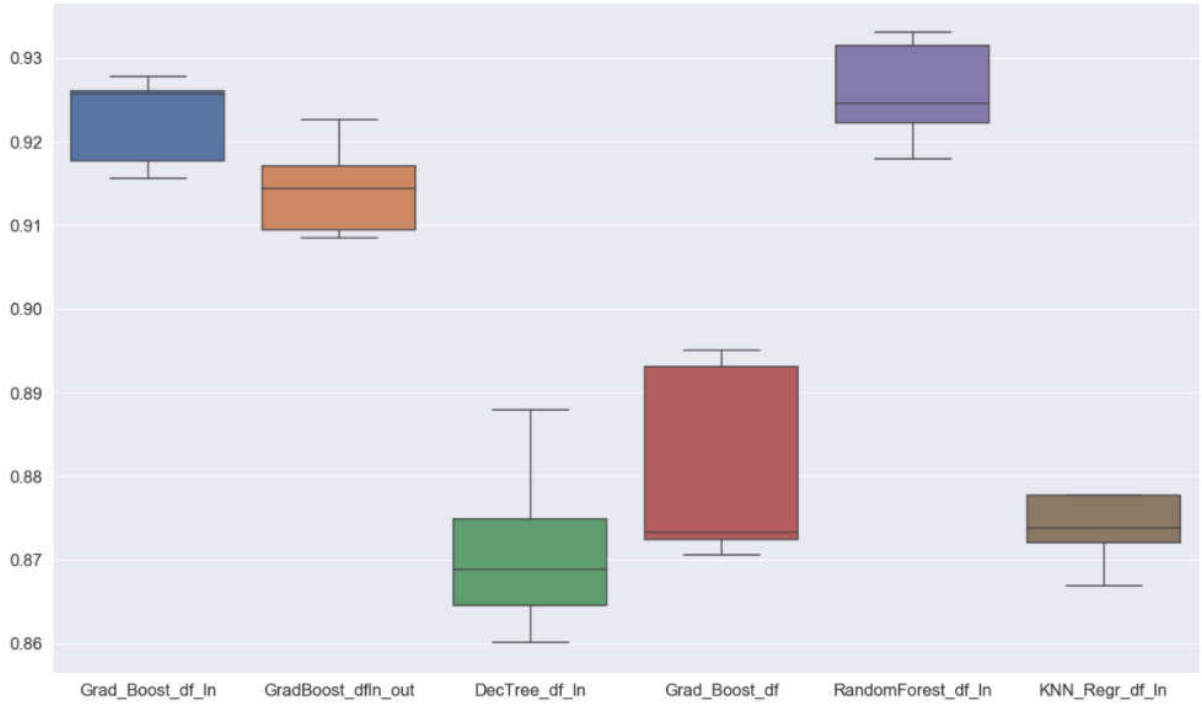
```
for function in ml_functions:
    for dataset,dataframe in datasets.items():
            modelos, scores_map,resultados = globals()[function](dataset,dataframe,modelos, scores_map, kf,rs)
            df_mod=pd.DataFrame(data = resultados)
            df_compara=pd.concat([df_compara,df_mod])
```

Avaliando o dataframe 'df_compara' e ordenando os respectivos modelos em ordem decrescente do R-quadrado conclui-se que os valores convertidos para escala logarítmica tiveram resultados melhores, especialmente aplicando o modelo *GradientBoostingRegres-sor*. Os valores de MAE e RMSE foram relativamente pequenos, indicando boa precisão dos melhores modelos. Também a diferença dos valores de MAE, RMSE e R-quadrado foram muito pequenas entre o conjunto de treino quanto de teste, indicando que os modelos prin-cipais aprenderam bem com os dados de treino e foram capazes de generalizar para os da-dos de teste.

Dataframe	MAE - Treina	MAE - Teste	RMSE - Treina	RMSE - Teste	R2 (Acurácia) - Treina	R2 (Acurácia) - Teste	
df_ln	0.11	0.14	0.15	0.19	96.04	93.43	GradientBoostingRegressor(alpha=(
df_ln_out	0.11	0.15	0.15	0.21	96.01	91.97	GradientBoostingRegressor(alpha=(
df	0.15	0.18	0.30	0.33	92.93	91.41	GradientBoostingRegressor(alpha=(
df_ln	0.00	0.16	0.02	0.23	99.95	90.24	KNeighborsRegressor(algorithm='b
df	0.00	0.19	0.02	0.37	99.95	89.54	KNeighborsRegressor(algorithm='b
df_ln	0.17	0.18	0.23	0.24	90.18	88.82	RandomForestRegressor(bootstrap=T
df_ln_out	0.00	0.17	0.02	0.25	99.95	88.50	KNeighborsRegressor(algorithm='b
df_ln	0.19	0.18	0.25	0.25	88.26	88.32	ElasticNet(alpha=0.0001, copy_X=T
df_ln	0.19	0.18	0.25	0.25	88.26	88.32	Ridge(alpha=0.0001, copy_X=True,

Segue o gráfico com o intervalo de acurácia calculados exibido para os melhores mo-delos classificados. Baixa variação indica que é um modelo razoavelmente robusto, mesmo que não seja o modelo de maior precisão. A transformação dos dados com distribuição as-

simétrica para escala logarítmica permitiu melhores resultados no geral, tanto com dataframe com *outliers* e sem *outliers*.



O modelo *GradientBoostingRegressor* foi bem classificado e funcionou bem em dados com distribuição simétrica. Percebe-se no mapa de *scores* essa diferença de amplitude dos intervalos entre dois dataframes em logaritmo natural e o que recebeu a padronização com *RobustScaler*.

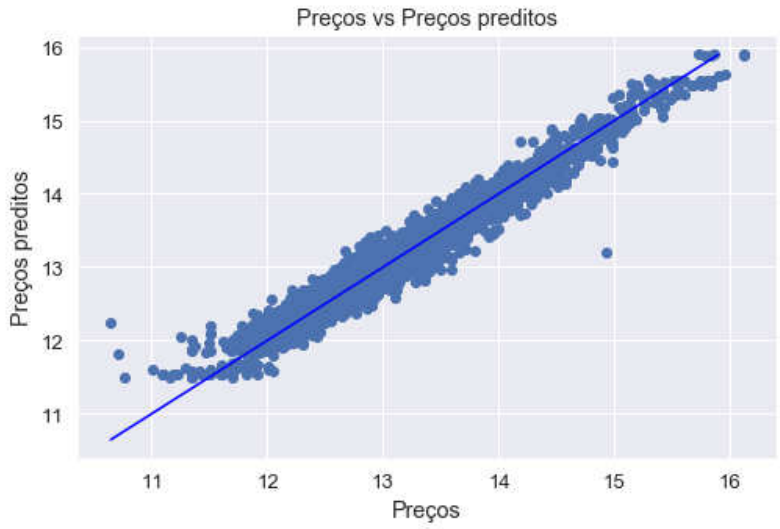
Detalhe importante para o modelo *RandomForestRegressor* aplicado ao dataframe "df_ln", esse modelo ficou classificado em quinto, mas no mapa de scores foi o modelo melhor posicionado e com pequena amplitude nos resultados.

Foram selecionados os modelos *GradientBoostingRegressor* e *RandonForestRegressor* para a análise de resíduos que será feita sobre os dados do dataframe "df_ln". Para isso será aplicado o modelo escolhido em todo o conjunto de dados do dataframe e com o resultado obtido serão plotados três gráficos.

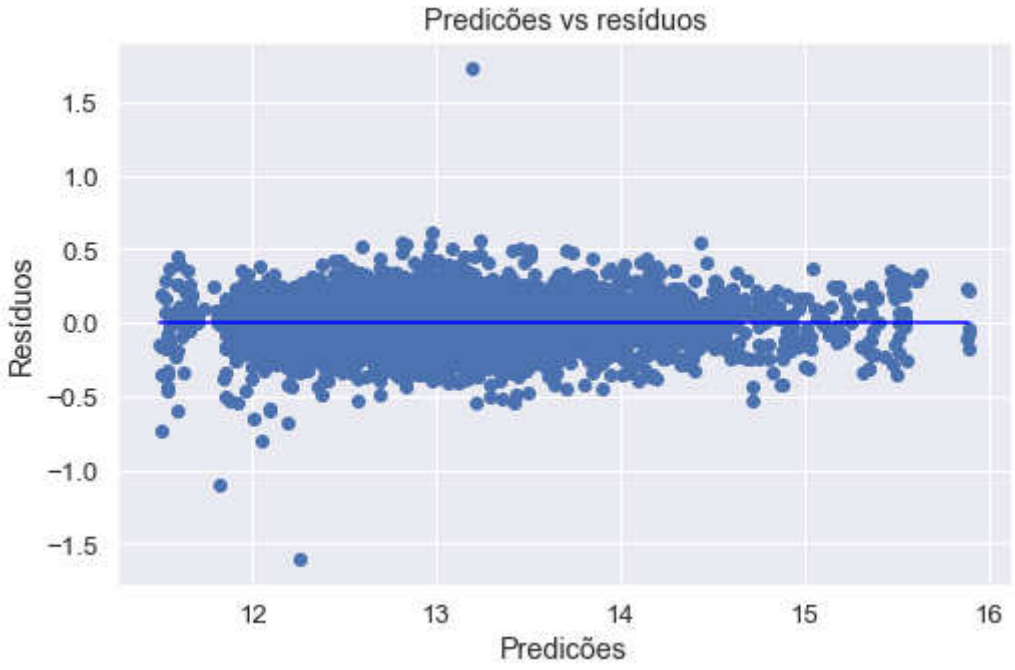
Após aplicar o modelo para todo o dataframe, foram calculadas as métricas (MAE, RMSE, e R^2) e aplicada a função “plotar_grafico()” que plotará os gráficos. O primeiro relacionando os preços e os preços preditos é desejável que o preço previsto acompanhe o preço real, assim, quanto mais preciso é o modelo, mais os pontos do gráfico se aglutinam em torno da linha azul formando um ângulo de 45°. O segundo relaciona as predições com os resíduos (preço – preço previsto), espera-se de um bom modelo preditivo que conforme os valores das predições aumentem, os valores dos resíduos permaneçam próximos de zero. Já o terceiro é um histograma com a distribuição de frequências dos resíduos, nele espera-se que seja uma distribuição normal centrada em zero.

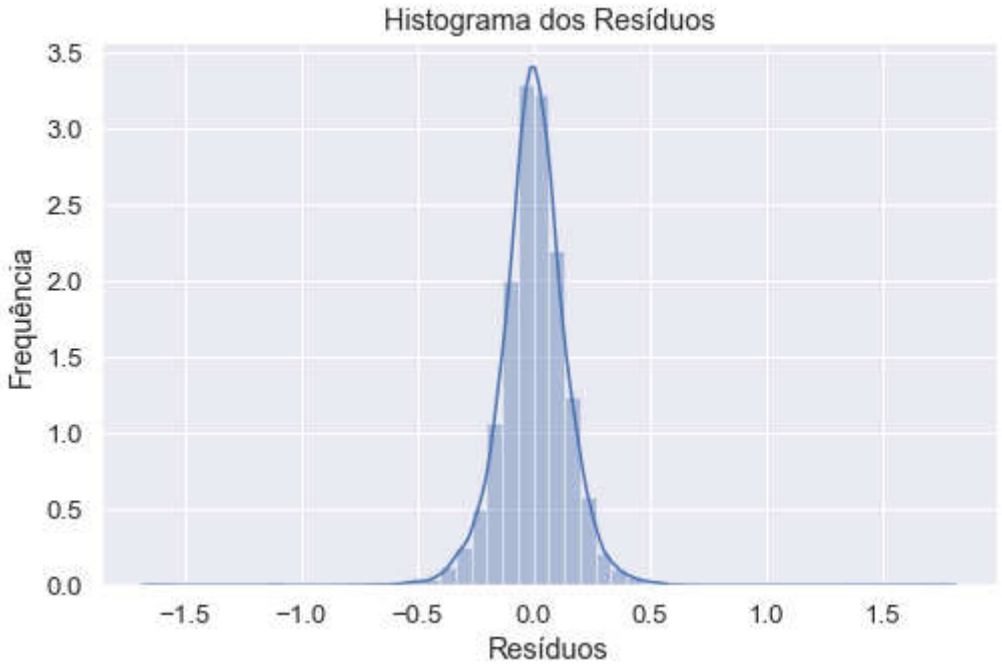
Abaixo segue o resultado da aplicação do modelo *GradientBoostingRegressor.* A acurácia do modelo para todo o conjunto de dados permanece no mesmo nível (96,25%) e os valores de MAE e RMSE também permanecem baixos. Percebe-se no gráfico de preços vs. preços preditos a tendência dos valores acompanharem a reta com ângulo de 45°. Mesmo havendo alguns pontos afastados, não prejudicou as métricas do modelo.

```
Dataframe: df_ln
Modelo: GradientBoostingRegressor
MAE:0.1041
RMSE: 0.1418
R²:0.9625
```



A seguir tem-se o gráfico de predições e resíduos e o histograma dos resíduos. No geral os valores de resíduos estão no intervalo de [-0,5 ; 0,5] tendo uma boa distribuição conforme avançam os valores de preço. Essa boa distribuição fica mais nítida ao olhar o histograma dos resíduos, a curva é acentuadamente leptocurtica.

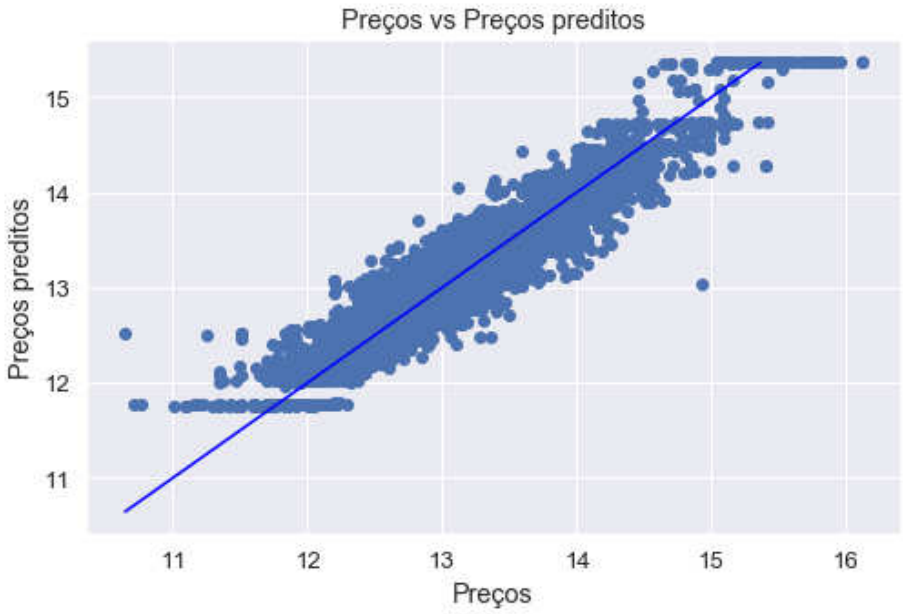


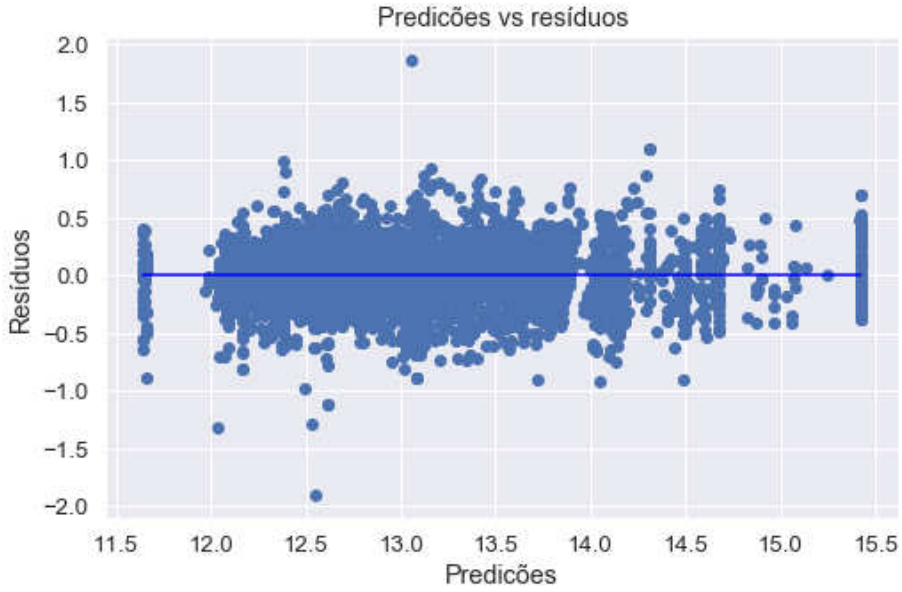


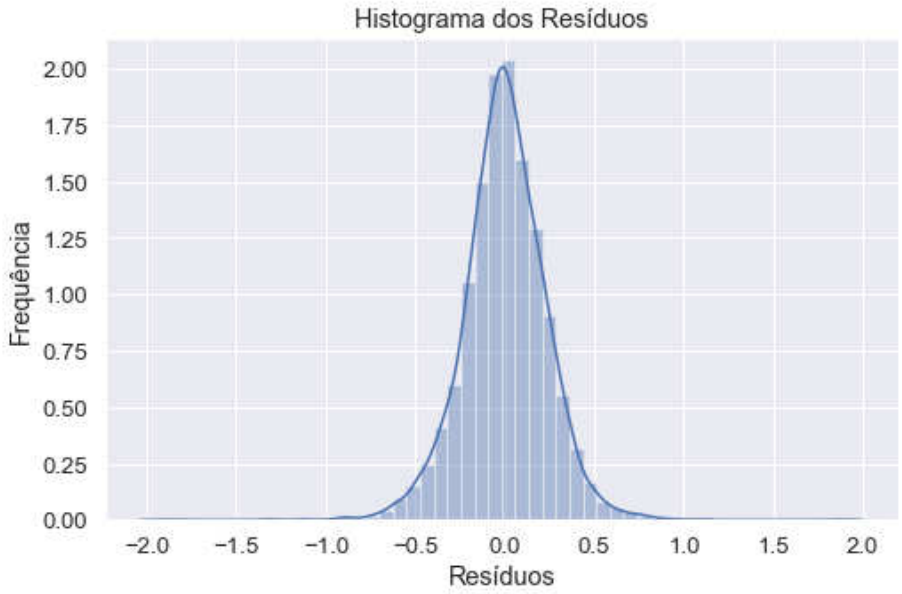
Por fim, segue o resultado da aplicação do modelo *RandonForestRegressor.* A acurácia do modelo para todo o conjunto de dados ficou em 90,6% e os valores de MAE e RMSE caíram levemente em comparação ao resultado da aplicação de todos os modelos. Percebe-se no gráfico de preços e preços preditos a tendência dos valores acompanharem a reta com ângulo de 45° a exceção dos valores extremos, tanto na base como no topo do gráfico há um achatamento dos valores de predição, ou seja, o modelo perde capacidade preditiva em relação a valores próximo dos extremos.

Na sequência tem-se o gráfico de predições e resíduos e o histograma dos resíduos. No geral, apesar dos valores de resíduos estarem no intervalo de [-0,5 ; 0,5] a distribuição não é muito uniforme conforme variam os preços. Existem alguns espaços vazios ao longo da reta, implicando que a distorção não está restrita aos valores extremos. O histograma dos resíduos mostra uma curva normal leptocurtica, mas com um intervalo de [-0,6; 0,6], um pouco maior que o histograma do outro modelo.

```
Dataframe: df_ln
Modelo: RandomForestRegressor
MAE:0.1668
RMSE: 0.2244
R²:0.906
```

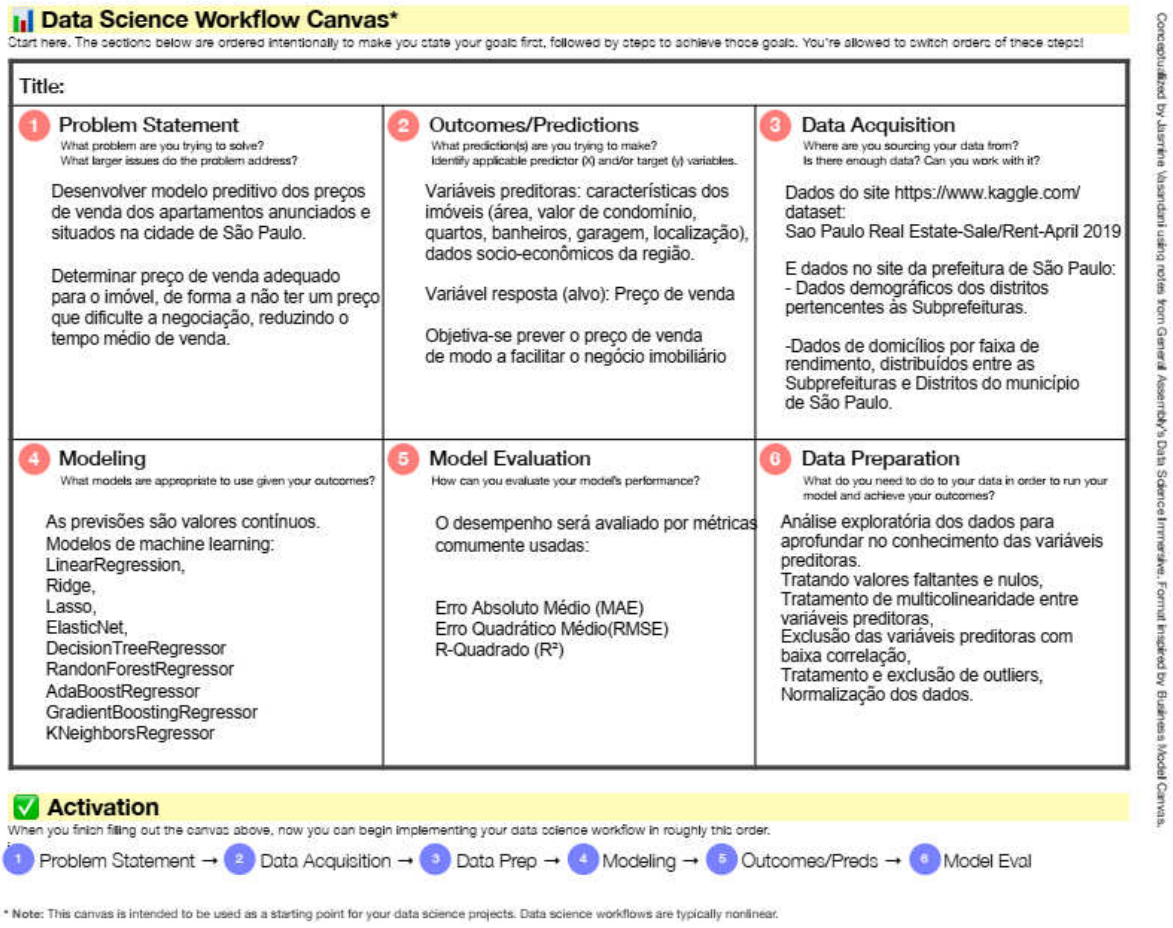



Predicões vs resíduos
Resíduos
Predicões
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
11.5
12.0
12.5
13.0
13.5
14.0
14.5
15.0
15.5


Histograma dos Resíduos
Frequência
Resíduos
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

7. Apresentação dos Resultados

Os resultados do projeto foram obtidos a partir do estabelecimento das metas mapeadas no modelo Canvas de Vasandani conforme a figura abaixo.



Data Science Workflow Canvas*

Start here. The sections below are ordered intentionally to make you state your goals first, followed by steps to achieve those goals. You're allowed to switch orders of these steps!

Title:

1 Problem Statement
What problem are you trying to solve?
What larger issues do the problem address?

Desenvolver modelo preditivo dos preços de venda dos apartamentos anunciados e situados na cidade de São Paulo.

Determinar preço de venda adequado para o imóvel, de forma a não ter um preço que dificulte a negociação, reduzindo o tempo médio de venda.

2 Outcomes/Predictions
What prediction(s) are you trying to make?
Identify applicable predictor (X) and/or target (y) variables.

Variáveis preditoras: características dos imóveis (área, valor de condomínio, quartos, banheiros, garagem, localização), dados socio-econômicos da região.

Variável resposta (alvo): Preço de venda

Objetiva-se prever o preço de venda de modo a facilitar o negócio imobiliário

3 Data Acquisition
Where are you sourcing your data from?
Is there enough data? Can you work with it?

Dados do site https://www.kaggle.com/dataset:
Sao Paulo Real Estate-Sale/Rent-April 2019

E dados no site da prefeitura de São Paulo:
- Dados demográficos dos distritos pertencentes às Subprefeituras.

-Dados de domicílios por faixa de rendimento, distribuídos entre as Subprefeituras e Distritos do município de São Paulo.

4 Modeling
What models are appropriate to use given your outcomes?

As previsões são valores contínuos.
Modelos de machine learning:
LinearRegression,
Ridge,
Lasso,
ElasticNet,
DecisionTreeRegressor
RandonForestRegressor
AdaBoostRegressor
GradientBoostingRegressor
KNeighborsRegressor

5 Model Evaluation
How can you evaluate your model's performance?

O desempenho será avaliado por métricas comumente usadas:

Erro Absoluto Médio (MAE)
Erro Quadrático Médio(RMSE)
R-Quadrado (R^2)

6 Data Preparation
What do you need to do to your data in order to run your model and achieve your outcomes?

Análise exploratória dos dados para aprofundar no conhecimento das variáveis preditoras.
Tratando valores faltantes e nulos,
Tratamento de multicolinearidade entre variáveis preditoras,
Exclusão das variáveis preditoras com baixa correlação,
Tratamento e exclusão de outliers,
Normalização dos dados.

Activation
When you finish filling out the canvas above, now you can begin implementing your data science workflow in roughly this order.

1 Problem Statement → 2 Data Acquisition → 3 Data Prep → 4 Modeling → 5 Outcomes/Preds → 6 Model Eval

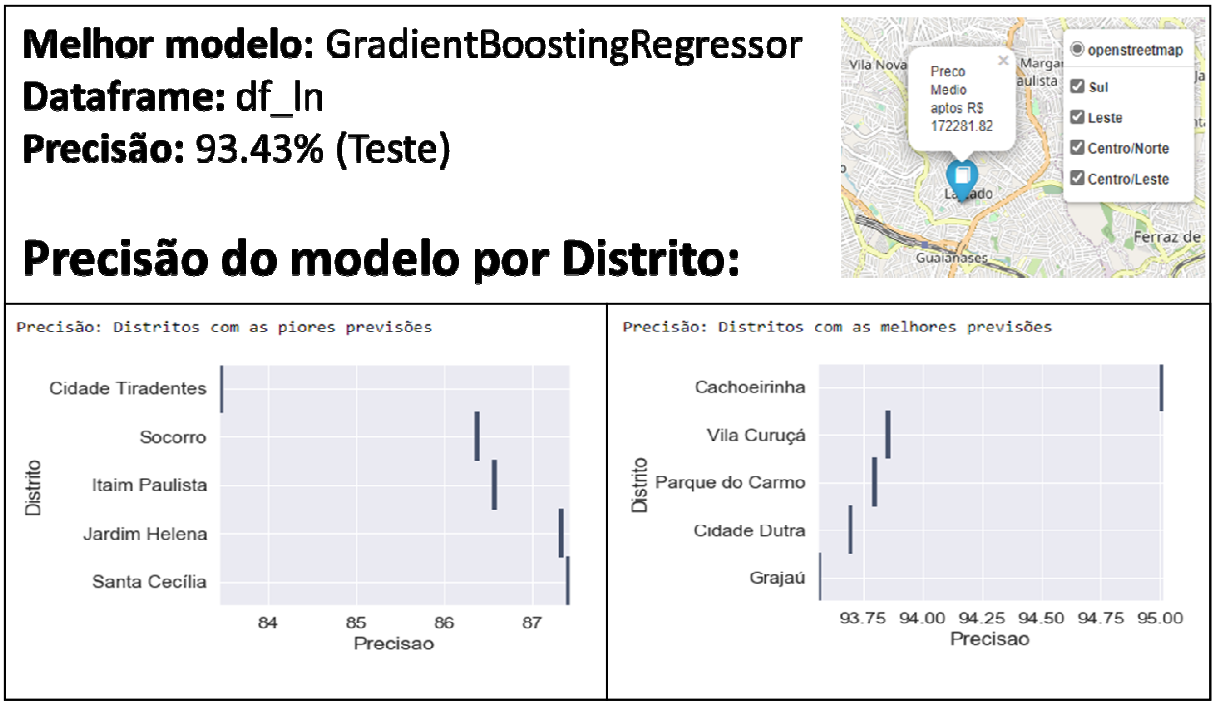
* Note: This canvas is intended to be used as a starting point for your data science projects. Data science workflows are typically nonlinear.

Conceptualized by Jasmine Vasandani using notes from General Assembly's Data Science Immersive. Format inspired by Business Model Canvas.

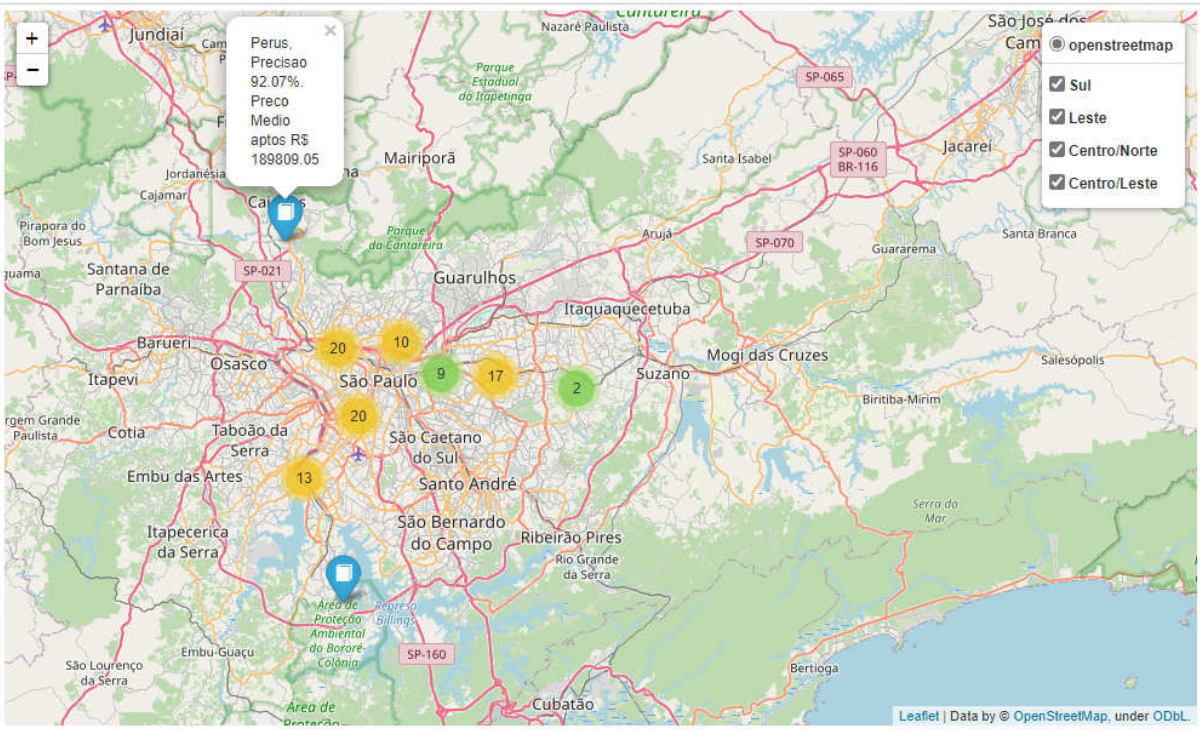
Dos modelos testados, conclui-se que o modelo de melhor desempenho foi o *GradientBoostingRegressor* aplicado ao dataframe "df_ln". O modelo apresentou os menores erro quadrático médio e erro absoluto médio na base de teste (RMSE igual a 0,19 e MAE igual a 0,14) e acurácia de 93,43% na base de teste.

O resultado obtido foi satisfatório uma vez que manteve um percentual de acurácia acima de 90%. Dessa forma, não invalida o uso como ferramenta para sugerir um preço de venda do imóvel. Também se um vendedor pretende vender a um preço e o modelo indicar

um valor menor, esse valor menor pode ser considerado como um preço mínimo a ser aceito pelo vendedor, caso contrário, o valor indicado pode ser considerado como novo valor de oferta. No caso de um comprador, se o modelo indicar um valor menor, ele pode considerar esse valor como objetivo de compra ou limite mínimo, inclusive para barganhar numa negociação; caso contrário (modelo indicando um valor maior que o imaginado), pode considerar esse preço como o máximo a ser pago. Segue imagem com apresentação de resultados:



Por fim, foi gerado através da biblioteca folium mapa de resultados em formato *.html* mostrando o percentual de precisão por distritos e o preço médio dos imóveis. A apresentação foi dividida por zonas (áreas) para facilitar pesquisas. O mapa é visualizado dentro do script ou pode ser visualizado via navegador de rede (*browser)*. Abaixo segue imagem demonstrativa do mapa gerado:


Perus, Precisao 92.07%. Preco Medio aptos R$ 189809.05
openstreetmap
Sul
Leste
Centro/Norte
Centro/Leste
Jundiaí
Nazaré Paulista
Mairiporã
Jordanésia
Cajamar
Pirapora do Bom Jesus
Santana de Parnaíba
SP-021
Guarulhos
Arujá
SP-065
SP-060
BR-116
Santa Isabel
Jacareí
Guararema
Santa Branca
SP-070
Itaquaquecetuba
Barueri
Osasco
Itapevi
São Paulo
Suzano
Mogi das Cruzes
Salesópolis
Biritiba-Mirim
Cotia
Taboão da Serra
Embu das Artes
São Caetano do Sul
Santo André
São Bernardo do Campo
Ribeirão Pires
Rio Grande da Serra
Serra do Mar
Itapecerica da Serra
Embu-Guaçu
São Lourenço da Serra
SP-160
Bertioga
Cubatão
20
10
9
17
2
20
13
Leaflet | Data by © OpenStreetMap, under ODbL.

8. Links

Link para o vídeo: https://youtu.be/Iiryjz9YOPw

Link para o repositório *github*: github.com/lmarchlopes/TCC-Puc-Minas--Preco_aptos-SP

REFERÊNCIAS

CALLEGARI-JACQUES, S. M. **Bioestatística: princípios e aplicações**. Traducao.[s.l.] Artmed Editora, 2009.

HEYDT, Michael. **Learning pandas Second edition** Birmingham: Packt, 2017.

THERESA, André Luís. **Predição da arrecadação de tributos federais, por município Brasileiro, a partir de informações sociais, econômicas, Demográficas e geopolíticas.** Trabalho de Conclusão de Curso(Especialização em Ciência de Dados e *Big Data*) – PUC-Minas. 2021.

Boosting Algorithms Explained, disponível em <https://towardsdatascience.com/boosting-algorithms-explained-d38f56ef3f30>. Acesso em 25 mar 2022.

Coeficiente de correlação de Pearson, disponível em <https://gpestatistica.netlify.app/blog/correlacao/>. Acesso em 10 mar 2022.

Decision Trees: Understanding the Basis of Ensemble Methods, disponível em <https://towardsdatascience.com/decision-trees-understanding-the-basis-of-ensemble-methods-e075d5bfa704>. Acesso em 25 mar 2022.

Dados estatísticos e informações econômicas da cidade de São Paulo, disponível em <https://www.prefeitura.sp.gov.br/cidade/secretarias/licenciamento/desenvolvimento_urbano/dados_estatisticos/info_cidade/economia/index.php?p=260269>. Acesso em 11 out 2021.

MAE, MSE, RMSE, Coefficient of Determination, Adjusted R Squared — Which Metric is Better?, disponível em <https://medium.com/analytics-vidhya/mae-mse-rmse-coefficient-of-determination-adjusted-r-squared-which-metric-is-better-cd0326a5697e>. Acesso em 25 mar 2022.

Modelos de Predição - Regressão de Ridge e Lasso, disponível em <https://medium.com/turing-talks/turing-talks-20-regress%C3%A3o-de-ridge-e-lasso-a0fc467b5629>. Acesso em 24 mar 2022.

Overfitting e Underfitting, disponível em <https://www.3dimensoes.com.br/post/overfitting-e-underfitting>. Acesso em 10 mar 2022.

Por que usar transformação logarítmica em dados?, disponível em <http://rstudio-pubs-static.s3.amazonaws.com/289147_99e32d5403f942339c3fe05414ac62fd.html>. Acesso em 22 fev 2022.

Quanto tempo leva para vender um imóvel?, disponível em <https://baggioimoveis.com.br/blog/2019/10/31/quanto-tempo-leva-para-vender-um-imovel/>. Acesso em 11 out 2021.

Random Forest: como funciona um dos algoritmos mais populares de ML, disponível em <https://medium.com/cinthiabpessanha/random-forest-como-funciona-um-dos-algoritmos-mais-populares-de-ml-cc1b8a58b3b4>. Acesso em 25 mar 2022.

Regularization Tutorial: Ridge, Lasso and Elastic Net, disponível em <https://www.datacamp.com/community/tutorials/tutorial-ridge-lasso-elastic-net>. Acesso em 25 mar 2022.

The Basics: KNN for classification and regression, disponível em <https://towardsdatascience.com/the-basics-knn-for-classification-and-regression-c1e8a6c955>. Acesso em 25 mar 2022.

Validação Cruzada Aninhada com Scikit-learn, disponível em <https://dataml.com.br/validacao-cruzada-aninhada-com-scikit-learn>. Acesso em 28 mar 2022.

Web Scraping: aquisição de dados em páginas web com Python, disponível em <https://medium.com/data-hackers/web-scraping-aquisi%C3%A7%C3%A3o-de-dados-em-p%C3%A1ginas-web-com-python-ec6e33e9e452>. Acesso em 10 mar 2022.

What is Gradient Boosting Regression?, disponível em <https://vitalflux.com/gradient-boosting-regression-python-examples/>. Acesso em 25 mar 2022.

Why Use Ensemble Learning?, disponível em <https://machinelearningmastery.com/why-use-ensemble-learning/>. Acesso em 25 mar 2022.

APÊNDICE

Programação/Scripts: Jupyter Notebook

1. 1 - Importando bibliotecas

In [ ]:

```
from zipfile import ZipFile

import requests
from bs4 import BeautifulSoup

import pandas as pd
import numpy as np

import folium
from geopy.geocoders import Nominatim
import seaborn as sns
import matplotlib.pyplot as plt
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator
from folium.plugins import MarkerCluster
from folium import plugins

from sklearn.preprocessing import RobustScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from math import sqrt
from scipy.stats import randint

# machine Learning
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn import linear_model
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import AdaBoostRegressor

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import RepeatedKFold


import warnings
warnings.filterwarnings("ignore")
```

1.2 Funções aplicadas neste notebook:

In [ ]:

```
def descompacta_zip(arq_descompactar):
    with ZipFile (arq_descompactar, 'r') as zip:
        zip.extractall()
        zip.printdir()
    return arq_descompactar+' Descompactado'


def extract_data_bairros(data):
    bairros = []
    for i in data:
        bairros.append(i.split('\n'))
    for i in bairros:
        if len(i)<4:
            bairros.remove(i)

    Distritos=[]
    Area=[]
    População=[]
    Densidade_Demográfica=[]
    for i in bairros:
        if i[-4].find('TOTAL')==-1:
            Distritos.append(i[-4])
            Area.append(i[-3])
            População.append(i[-2])
            Densidade_Demográfica.append(i[-1])

    len(Distritos),len(Area),len(População),len(Densidade_Demográfica)
    if len(Distritos)==len(Area)==len(População)==len(Densidade_Demográfica):
        n = len(Distritos)
        print('Extração de dados está CORRETA\nCada lista possui',n, 'elementos')

    else:
        print('Extração de dados NÃO está correta!')

    # Criação do Dataframe

    colunas = ['Distrito', Area[0],População[0],Densidade_Demográfica[0],]

    df = pd.DataFrame(list(zip(Distritos, Area,População,Densidade_Demográfica)), colum
ns = colunas)
    df.drop(index=0, inplace=True)
    return df


#Função converte colunas numéricas de um Dataframe para logaritno natural: Precisa info
rmar colunas
def simetric(df, cols):
    cols_ln=[]
    for col in cols:
        sk = df[col].skew()
        if (sk>1 or sk<-1):
            print('Assimetria (Skewness) em %s é %.2f' %(col,sk))
            nova_coluna = col+'_ln'
            df[nova_coluna]=(np.log(df[col])).replace(-np.inf, 0)
            sk_ln = df[nova_coluna].skew()
            print('Assimetria (Skewness) em %s é %.2f' %(nova_coluna,sk_ln))
            if sk_ln > sk:
                df.drop(columns = nova_coluna, inplace = True)
                print('Coluna %s Não alterada\n' %(col))
```

```
            else:
                df[col]=df[nova_coluna]
                df.drop(columns = nova_coluna, inplace = True)
#                cols_ln.append(nova_coluna)

                print('Coluna %s ajustada \n' %(col))


    return cols_ln


# Função para calcular limites máximos e mínimos para outliers
def outlier_iqr(df, cols):
    for i in cols:
        df.sort_values(by=i, ascending=True, na_position='last')
        q1, q3 = np.nanpercentile(df[i], [25,75])
        iqr = q3-q1
        lower_bound = q1-(1.5*iqr)
        upper_bound = q3+(1.5*iqr)
        outlier_data = df[i][(df[i] < lower_bound) | (df[i] > upper_bound)] #creating a
series of outlier data
        perc = (outlier_data.count()/df[i].count())*100
        print('Outliers em %s %.2f%% Total de registros: %.f' %(i, perc, outlier_data.c
ount()))
    return lower_bound, upper_bound



#Cria um scatter plot com o resultado do modelo de machine learning.
def plotar_grafico(y_t,y_pred):
    # Visualizando as diferenças entre preços atuais e preços preditos
    sns.set(font_scale = 1.2)
    plt.figure(figsize = (8,5))
    plt.scatter(y_t, y_pred)
    range = [y_t.min(), y_pred.max()]
    plt.plot(range, range, 'blue')
    plt.xlabel("Preços")
    plt.ylabel("Preços preditos")
    plt.title("Preços vs Preços preditos")
    plt.show()

    # Checando resíduos
    sns.set(font_scale = 1.2)
    plt.figure(figsize = (8,5))
    plt.scatter(y_pred,y_t-y_pred)
    plt.plot([y_pred.min(),y_pred.max()],[0,0], 'blue')
    plt.title("Predicões vs resíduos")
    plt.xlabel("Predicões")
    plt.ylabel("Resíduos")
    plt.show()

    # Checando normalidade dos erros
    sns.set(font_scale = 1.2)
    plt.figure(figsize = (8,5))
    sns.distplot(y_t-y_pred)
    plt.title("Histograma dos Resíduos")
    plt.xlabel("Resíduos")
    plt.ylabel("Frequência")
    plt.show()
    return
```

Coleta dos dados

Primeiro Dataframe - df_imoveis

In [ ]:

```
# Chamando função para descompactar .zip
descompacta_zip('sao-paulo-properties-april-2019.csv.zip')
```

In [ ]:

```
# Abertura de arquivo e visualização dos primeiros registros:

csv_file = 'sao-paulo-properties-april-2019.csv'

df = pd.read_csv(csv_file, sep=',')

del csv_file
display(df.head())
df.info()
```

In [ ]:

```
# Checando valores únicos para tipo de negociação (será extraído apenas os registros de venda)

print('Tipo de negociação:', list(df['Negotiation Type'].unique()))
```

In [ ]:

```
# Seleção dos imóveis para venda:
df_imoveis = df.loc[df['Negotiation Type']== 'sale']
df_imoveis.drop(columns=['Negotiation Type'], inplace=True)

# Exclusão de dados duplicados, renomeação das colunas e atualização do index
df_imoveis = df_imoveis.drop_duplicates()
df_imoveis.rename(columns={'Price': 'Preco', 'Size':'Area', 'Rooms': 'Quartos', 'Toilets':'Banheiros',
                           'Parking': 'Garagem', 'Elevator': 'Elevador', 'Furnished': 'Mobiliado', 'Swimming Pool': 'Piscina',
                           'New': 'Novo', 'District': 'Distrito', 'Property Type':'Tipo_imovel'}, inplace=True)

# Como verificado, a coluna "Distrito" contém nome do distrito e cidade.
# Fatiamento dessa coluna em 'Distrito' e 'Cidade':
df_imoveis[['Distrito','Cidade']] = df_imoveis['Distrito'].str.split('/', expand = True)
df_imoveis.reset_index(drop=True, inplace=True)

# Verificação se todos os registros são da cidade de São Paulo e quais os tipos de imóveis que compões o DataFrame:
print(' Cidades no dataframe:', list(df_imoveis['Cidade'].unique()), '\n',
      'Tipos de imóveis: ', list(df_imoveis['Tipo_imovel'].unique()), '\n')

del df
df_imoveis.info()
```

In [ ]:

```
# Como os registros são únicos, essas colunas serão excluídas

df_imoveis.drop(columns=['Cidade','Tipo_imovel'], inplace=True)
df_imoveis.head()
```

In [ ]:

```
pd.options.display.float_format = '{:.4f}'.format
```

In [ ]:

```
# Descrição sumária dos registros do Dataframe:
display(df_imoveis.round().describe())
# Checagem de valores Nan:
print('\n',"Total de valores NaN: ", df_imoveis.isna().sum().sum())
```

In [ ]:

```
# Visualização da distribuição dos dados do Dataframe

df_hist = df_imoveis[['Preco', 'Condo', 'Area', 'Quartos', 'Banheiros', 'Suites', 'Garagem']]

fig, axs = plt.subplots(ncols=2, nrows=4, figsize=(12, 8))
index = 0
axs = axs.flatten()
for k,v in df_hist.items():
    sns.histplot(v, ax=axs[index])
    index += 1
plt.tight_layout(pad=0., w_pad=0.5, h_pad=5.0)

del df_hist
```

Segundo Dataframe - df_bairros

In [ ]:

```
# Função para buscar dados dos bairros e subprefeituras da cidade de São Paulo:
# <'https://www.prefeitura.sp.gov.br/cidade/secretarias/subprefeituras/subprefeituras/dados_demograficos/index.php?p=12758'>

# Uso de Beautiful Soup

url = 'https://www.prefeitura.sp.gov.br/cidade/secretarias/subprefeituras/subprefeituras/dados_demograficos/index.php?p=12758'
html_text = requests.get(url).text
soup = BeautifulSoup(html_text, 'html.parser')
title = soup.find('tbody').get_text()
data=title.strip().split('\n\n')
data
```

In [ ]:

```
# Criação do Dataframe df_bairros

df_bairros = extract_data_bairros(data)

df_bairros.to_csv("df_bairros.csv")

type(df_bairros)
```

In [ ]:

```
#df_bairros = pd.read_csv('bairros_SP.csv', index_col =0)
df_bairros.info()
```

In [ ]:

```
# Criação do Dataframe df_bairros
display(df_bairros.head())
display(df_bairros.tail())
```

In [ ]:

```
# Uso de função lambda e .replace para mudar vírgulas por pontos e pontos e vírgulas po
r espaço vazio

df_bairros['Área (km²)'] = df_bairros['Área (km²)'].apply(lambda x: float(x.replace(","
,".")))
df_bairros['População (2010)'] = df_bairros['População (2010)'].apply(lambda x: float(x
.replace(".","")))
df_bairros['Densidade Demográfica (Hab/km²)'] = df_bairros['Densidade Demográfica (Hab/
km²)'].apply(lambda x: float(x.replace(",","").replace(".","")))

# Visualizando as 'pontas' do Dataframe
display(df_bairros.head())
display(df_bairros.tail())
```

In [ ]:

```
##df_bairros = pd.read_csv('bairros_SP.csv', index_col=0)
```

In [ ]:

```
# Descrição sumária do Dataframe:
display(df_bairros.info())
display(df_bairros.describe())
print('\n',"Valores NaN:",'\n\n', df_bairros.isna().sum())
```

In [ ]:

```
# Visualização da distribuição dos dados do Dataframe
df= df_bairros[['Área (km²)','População (2010)', 'Densidade Demográfica (Hab/km²)']]

fig, axs = plt.subplots(ncols=3, nrows=1, figsize=(20, 5))
index = 0
axs = axs.flatten()
for k,v in df.items():
    sns.histplot(v, ax=axs[index])
    index += 1
plt.tight_layout(pad=0.2, w_pad=0.3, h_pad=3.0)
```

Terceiro Dataframe - df_rendas

In [ ]:

```
# Abrindo arquivo .xlsx e renomeando colunas

df_rendas = pd.read_excel('Domicilios_faixa_rendimento_sal_minimos_2010.xls', skiprows=
6, skipfooter=5)

df_rendas.rename(columns = {'Unnamed: 0' : 'Distrito', 'Unnamed: 1': 'Total_domicilios'
,
                            "Sem rendimento (3)": "Sem_rendimento"}, inplace=True)

# Renomeando a última coluna:
cols = df_rendas.columns
for col in cols:
    if col.startswith('Sem'):
        df_rendas.rename(columns = {col:"Sem_rendimento"}, inplace=True)
df_rendas.columns
```

In [ ]:

```
# Visualizando as 'pontas' do Dataframe
display(df_rendas.head())
display(df_rendas.tail())
```

In [ ]:

```
# Descrição sumária do Dataframe:
display(df_rendas.info())
display(df_rendas.describe())
df_rendas.isna().sum()
```

In [ ]:

```
#Correção Registro Distrito 'São Miguel' e 'Moóca'

df_rendas.iloc[[105],[0]]='São Miguel'

df_rendas.loc[df_rendas['Distrito'] == 'Moóca', 'Distrito']='Mooca'
```

In [ ]:

```
# Exclusão de registros manualmente.
#Ex.: 2 Registros Butantã: um é o distrito e o outro se refere a Subprefeitura
index_drop= [0,1,5,11,15,19,23,28,31,34,37,41,44,49,51,54,61,64,71,74,79,82,87,91,95,99
,103,107,116,120,124]

df_rendas.drop(index=index_drop, inplace=True)
df_rendas.drop_duplicates(inplace=True)
# Correção de pontos e vírgulas para números e conversão para percentagem
cols = df_rendas.columns

for col in cols:
    if (col != 'Distrito'):
        df_rendas[col] = df_rendas[col].apply(lambda x: float(x.replace(".","")))

print("Total de registros df_rendas: ",len(list(df_rendas['Distrito'])))
```

In [ ]:

```
# Conversão para percentagem

for col in cols:
    if (col != 'Distrito') and (col != 'Total_domicilios'):
        df_rendas[col+'_SM_%'] = round(df_rendas[col]/df_rendas['Total_domicilios']*100
,2)
        df_rendas.drop(columns=col, inplace=True)
```

In [ ]:

```
# Descrição sumária do Dataframe e checagem de valores Nan:
display(df_rendas.info())
display(df_rendas.describe())
print("Total de valores NaN: ", df_rendas.isna().sum().sum())
```

Primeira junção de Dataframes:

df_bairros e df_rendas

In [ ]:

```
# Merge dos Dataframes df_bairros e df_rendas, inner join e contagem do número de regis
tros
df_distritos = pd.merge(df_bairros, df_rendas, on='Distrito', how='inner')
print(' Total de registros em df_distritos: ', len(df_distritos), '\n',
      'Total de registros em df_bairros:   ',len(df_bairros),'\n',
      'Total de registros em df_rendas:   ',len(df_rendas))

# Checar Nan's nas colunas e percentual de Nan's em cada coluna
print(" Total de Valores NaNs:               ", df_distritos.isna().sum().sum())
```

In [ ]:

```
display(df_distritos.head())
df_distritos.tail()
```

Segunda junção de Dataframes:

df_imoveis e df_distritos

In [ ]:

```
# Merge Dataframes df_imoveis e df_distritos e posterior verificação pois a chave "Dist
rito" em df_imoveis pode conter erros

df = pd.merge(df_imoveis, df_distritos, on = 'Distrito', how = 'left')
l_distr = df['Distrito'].loc[df['População (2010)'].isna()].unique()
print(' Total de Distritos não identificados:', len(l_distr),'\n','Distritos não identi
ficados: ', l_distr)
```

In [ ]:

```
# Correção gramatical dos nomes dos distritos

df_distritos.loc[df_distritos['Distrito'] == 'Guaianases', 'Distrito']='Guaianazes'

df_imoveis.loc[df_imoveis['Distrito'] == 'Jardim São Luis', 'Distrito']='Jardim São Luí
s'
df_imoveis.loc[df_imoveis['Distrito'] == 'Medeiros', 'Distrito']='Vila Medeiros'

# correção de lugares:df_imoveis com local identificado pelo nome da vila ou bairro e d
f_distritos pelo nome oficial
df_imoveis.loc[df_imoveis['Distrito'] == 'Vila Madalena', 'Distrito']='Pinheiros'
df_imoveis.loc[df_imoveis['Distrito'] == 'Brooklin', 'Distrito']='Itaim Bibi'
df_imoveis.loc[df_imoveis['Distrito'] == 'Vila Olimpia', 'Distrito']='Itaim Bibi'
```

In [ ]:

```
# Merge com adequação dos nomes dos Distritos

df = pd.merge(df_imoveis, df_distritos, on = 'Distrito', how = 'left')

l_distr = df['Distrito'].loc[df['População (2010)'].isna()].unique()
print(' Total de Distritos não identificados:', len(l_distr),'\n','Distritos não identi
ficados: ', l_distr)

df.dropna(inplace=True)
df.reset_index(drop = True,inplace=True)

print( " Total de valores nulos:", df.isna().sum().sum(),'\n Dimensão df:', df.shape)
```

In [ ]:

```
# Deletando demais dataframes:

del df_bairros
del df_rendas
del df_imoveis
#del df_distritos
del url
del html_text
del soup
del title
del data
```

In [ ]:

```
df.to_csv('df_SP_28-03-22.csv')
```

Análise Exploratória de Dados

In [ ]:

```
df.info()
```

In [ ]:

```
# Visualização da distribuição dos dados do Dataframe

l_columns = list(df.columns[0:7])+ list(df.columns[14:])
sns.set(font_scale = 1.5)
hist = df[l_columns].hist(bins=15, figsize=(15,20))
```

In [ ]:

```
# Continuando a exploração dos dados: Relação do Preço com as variáveis: elevador, mobi
liado, piscina e novo
l_columns = df.columns[7:11]
pos = 0
plt.figure(figsize = (10,10))
for i in l_columns:
    pos +=1
    plt.subplot(2,2,pos)
    ax = sns.boxenplot(x = i , y = 'Preco',data = df )
```

In [ ]:

```
# Visualização Relação de Preço do imóvel em relação à: Condomínio, Área, Quartos, Banh
eiros, Suítes e Garagem
pd.options.display.float_format = '{:.2f}'.format
l_columns = df.columns[1:7]
sns.set(font_scale = 1.2)
pos = 0
plt.figure(figsize = (15,8))
for i in l_columns:
    pos +=1
    plt.subplot(2,3,pos)
    ax = sns.regplot(x = i, y = 'Preco', data = df)
```

In [ ]:

```
# Continuando a exploração dos dados: Relação do Preço com as demais variáveis advindas
dos dataset df_distritos:

l_columns = df.columns[14:]
pos = 0
plt.figure(figsize = (14,16))
for i in l_columns:
    pos +=1
    plt.subplot(4,3,pos)
    ax = sns.regplot(x = i, y = 'Preco', data = df, x_bins=50)
```

In [ ]:

```
# Variáveis de localização: Latitude e Longitude
# É esperado que formem um desenho
ax = sns.pairplot(df, y_vars = 'Latitude', x_vars = ['Longitude'],
                  height = 4, kind = 'reg')

ax.fig.suptitle('Dispersão entre as Variáveis', fontsize=10, y=0.5)
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df.groupby('Distrito')['Preco'].mean().reset_index()

grouped.rename(columns={'Preco':'Preco_Medio'}, inplace=True)

grouped = grouped.sort_values(by = 'Preco_Medio', ascending = False).head(5)
plt.figure(figsize = (5,4))
sns.set(font_scale = 1.40)
ax = sns.barplot(x='Preco_Medio', y='Distrito', data = grouped,palette = 'plasma')
print(" Distritos com os maiores preço médio de apartamentos: ")
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df.groupby('Distrito')['Preco'].mean().reset_index()


grouped.rename(columns={'Preco':'Preco_Medio'}, inplace=True)

grouped = grouped.sort_values(by = 'Preco_Medio', ascending = True).head(5)
plt.figure(figsize = (5,4))
sns.set(font_scale = 1.40)
ax = sns.barplot(x='Preco_Medio', y='Distrito', data = grouped,palette = 'plasma')
print(" Distritos com os menores preço médio de apartamentos: ")
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df.groupby('Distrito')['Preco'].mean().reset_index()

grouped = grouped.loc[grouped['Preco']>1000000].sort_values(by = 'Preco',ascending = Fa
lse)
plt.figure(figsize = (10,10))
sns.set(font_scale = 1.4)
ax = sns.barplot(x='Preco', y='Distrito', data = grouped,palette = 'plasma')
print("Total de distritos com preço médio acima de 1 milhão de reais: ", len(grouped))
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df.groupby('Distrito')['Preco'].mean().reset_index()

grouped = grouped.loc[grouped['Preco']<250000].sort_values(by = 'Preco',ascending = Fal
se)
plt.figure(figsize = (10,10))
sns.set(font_scale = 1.4)
ax = sns.barplot(x='Preco', y='Distrito', data = grouped,palette = 'plasma')
print("Total de distritos com preço médio abaixo de 250 mil de reais: ", len(grouped))
```

In [ ]:

```
# Dicionário para gerar Nuvem de Palavras (worldcloud)

#from PIL import Image
# Import image to np.array
#mask = np.array(Image.open('SP_mapas.png'))

data_cloud = dict(df['Distrito'].value_counts())

# gerar uma wordcloud
wordcloud = WordCloud(background_color="white", colormap = "Greens_r",
                      width=1600, height=1000, max_words=93,
                      max_font_size=250,
                      min_font_size=3).generate_from_frequencies(data_cloud)

# mostrar a imagem final
fig, ax = plt.subplots(figsize=(14,14))
ax.imshow(wordcloud, interpolation='bilinear')
ax.set_axis_off()

plt.imshow(wordcloud);
#wordcloud.to_file("imóveis_summary.png")
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df.groupby('Distrito')['População (2010)'].mean().reset_index()

grouped = grouped.loc[grouped['População (2010)']>200000].sort_values(by = 'População
 (2010)',
                                                                       a
scending = False)
plt.figure(figsize = (8,12))
sns.set(font_scale = 3.39)
ax = sns.barplot(x='População (2010)', y='Distrito', data = grouped,palette = 'plasma')
print("Total de distritos com Densidade Demográfica (Hab/km²) abaixo de 5000 hab/km²: "
, len(grouped))
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df.groupby('Distrito')['Mais de 20_SM_%'].mean().reset_index()

grouped = grouped.loc[grouped['Mais de 20_SM_%']>25].sort_values(by = 'Mais de 20_SM_%'
,
                                                                 a
scending = False)
plt.figure(figsize = (8,12))
sns.set(font_scale = 3.39)
ax = sns.barplot(x='Mais de 20_SM_%', y='Distrito', data = grouped,palette = 'plasma')
print("Total de distritos com Densidade Demográfica (Hab/km²) abaixo de 5000 hab/km²: "
, len(grouped))
```

In [ ]:

```
# Uso biblioteca folium usando colunas Latitudes e Longitudes informadas:

#from folium.plugins import MarkerCluster

m = folium.Map(
    location=[-23.3, -46.6],
    tiles='Stamen Toner',
    zoom_start=8
)
mc = MarkerCluster()

for index, value in df.iterrows():
  mc.add_child(folium.Marker([value['Latitude'], value['Longitude']],
              popup=str(value['Distrito']),
              tooltip=value['Distrito'],
              icon=folium.Icon(icon='book'))).add_to(m)
m
```

Tratamento dados NaN

Tratamento Valor de condomínio

In [ ]:

```
# A princípio não há dados nulos

print('Total de dados nulos: ', df.isna().sum().sum())
```

In [ ]:

```
# Visualizando valores muito baixos, há valores 0 para condomínio.
# E valores muito próximos de Zero

cond_0 = len(df[df['Condo']==0])

prox_0 = len(df[df['Condo']<=10].loc[df['Condo']>0])

print('Total de registros com Condomínio igual a zero: ', cond_0)

print('Total de registros com Condomínio próximo a zero: ', prox_0)
```

In [ ]:

```
# Substituindo valores iguais ou menores que 10 por None:

df['Condo'] = np.where(df['Condo']<= 10,np.nan, df['Condo'])

# substituir o missing pela mediana da coluna
df['Condo'].fillna(round(df['Condo'].median()), inplace=True)


df.Condo.describe()
```

In [ ]:

```
df['Condo'] = df['Condo'].astype('int64')
```

Tratamento Valores de Latitude e Longitude

In [ ]:

```
# Verificando valores iguais a zero:

lat_zero = len(df[['Distrito','Latitude', 'Longitude']].loc[df['Latitude']==0])
long_zero = len(df[['Distrito','Latitude', 'Longitude']].loc[df['Longitude']==0])

print('Total de registros de Latitude iguais a zero: ',lat_zero)
print('Total de registros de Longitude iguais a zero: ',long_zero)
```

In [ ]:

```
# Uso Biblioteca Nominatim para identificar a Latitude e Longitude Máxima e Mínima da c
idade de São Paulo

place = 'São Paulo, Região Imediata de São Paulo, Região Metropolitana de São Paulo,'

geolocator = Nominatim(user_agent="geolocalização")
d_lat_sp={}
d_long_sp={}
location = geolocator.geocode( place)
d_lat_sp['São Paulo']=[float(location.raw['boundingbox'][0]),float(location.raw['boundi
ngbox'][1])]
d_long_sp['São Paulo']= [float(location.raw['boundingbox'][2]), float(location.raw['bou
ndingbox'][3])]

print('Latitudes Mínima e Máxima:', d_lat_sp)
print('Longitudes Mínima e Máxima:',d_long_sp )
```

In [ ]:

```
# Substituindo os registros de posição geográfica incorretos por NaN(fora dos limites m
áximos e mínimos)

df['Latitude'] = np.where(df['Latitude']>=d_lat_sp['São Paulo'][0],
                                          np.where(df['Latitude']<=d_lat_sp['São Paulo'
][1],
                                         df['Latitude'],
                                          np.nan),np.nan)

df['Longitude'] = np.where(df['Longitude']>=d_long_sp['São Paulo'][0],
                                          np.where(df['Longitude']<=d_long_sp['São Paul
o'][1],
                                         df['Longitude'],
                                          np.nan),np.nan)
```

In [ ]:

```
# A partir do df, cria outros dois dataframes com as médias das latitudes e longitudes
 para cada distrito.
# Com merge cria-se novo df (df_medias) com as medias e elimina-se os valor nulos
# Assim obtêm-se a média de latitudes e longitudes em cada distrito para substituir os
 registros nulos:

df_media_lat = df[['Distrito', 'Latitude']].groupby(['Distrito']).mean().reset_index()
df_media_long = df[['Distrito', 'Longitude']].groupby(['Distrito']).mean().reset_index
()

df_medias = pd.merge(df_media_lat, df_media_long, on = 'Distrito', how = 'inner')
df_medias.dropna(inplace=True)
print('Dimensão do DataFrame df_medias: ', df_medias.shape)
display(df_medias.head())
```

In [ ]:

```
# Verificando quais os distritos não contêm posição geográfica subtraindo-os do df_medi
as no df,
# resultando numa relação de distritos sem as médias das posições geográficas.

places = list(set(df['Distrito'].unique())-set(df_medias['Distrito'].unique()))
print( "Relação de Distritos sem valores de Latitude e Longitude:\n\n", places)
```

In [ ]:

```
# Com a biblioteca Nominatim, encontra-se Latitude e Longitude e insere no df_medias:

places = list(set(df['Distrito'].unique())-set(df_medias['Distrito'].unique()))

geolocator = Nominatim(user_agent="geolocalização")
for place in places:
    location = geolocator.geocode( place+'- São Paulo - SP')
    df_medias = df_medias.append({'Distrito' : place , 'Latitude' : location.latitude,
                                  'Longitude' : location.longitude}, ignore_index=True)

df_medias.tail()
```

In [ ]:

```
# No dataframe df, substitui as latitudes e longitudes incorretas pelo nome do respecti
vo distrito
df_medias.rename(columns={'Latitude':'media_Latitude', 'Longitude': 'media_Longitude'},
inplace=True)
df = pd.merge(df, df_medias, on='Distrito', how='left')

df["Latitude"].fillna(df['media_Latitude'], inplace=True)
df["Longitude"].fillna(df['media_Longitude'], inplace=True)

df.drop(columns = ['media_Latitude', 'media_Longitude'], inplace = True)
print('Total registros de Latitude:', df.Latitude.count())
print('Total registros de Longitude:', df.Longitude.count())
```

In [ ]:

```
m = folium.Map(
    location=[-23.5, -46.6],
    tiles='Stamen Toner',
    zoom_start=8
)
mc = MarkerCluster()

for index, value in df.iterrows():
    mc.add_child(folium.Marker([value['Latitude'], value['Longitude']],
              popup=str(value['Distrito']),
              tooltip=value['Distrito'],
              icon=folium.Icon(icon='book'))).add_to(m)
m
```

In [ ]:

```
coordenadas=[]
for lat,lng in zip(df.Latitude.values,df.Longitude.values):
    coordenadas.append([lat,lng])
m = folium.Map(location=[-23.55,-46.63], zoom_start=10)
m.add_child(plugins.HeatMap(coordenadas))
m
```

In [ ]:

```
lat_zero = len(df[['Distrito','Latitude', 'Longitude']].loc[df['Latitude']==0])
long_zero = len(df[['Distrito','Latitude', 'Longitude']].loc[df['Longitude']==0])

print('Total de registros de Latitude iguais a zero:  ',lat_zero)
print('Total de registros de Longitude iguais a zero: ',long_zero)
```

In [ ]:

```
sns.set(font_scale = 1.5)
ax = sns.pairplot(df, y_vars = 'Latitude', x_vars = ['Longitude'],
                  height = 5, kind = 'reg')

ax.fig.suptitle('Dispersão entre as Variáveis', fontsize=2, y=0.5)
```

In [ ]:

```
# Exclusão de dados duplicados

df = df.drop_duplicates()
df.reset_index(drop=True, inplace=True)
display(df.shape)
```

In [ ]:

```
df_orig = df.copy()
df_orig.to_csv("dataset_consolidado.csv")
```

In [ ]:

```
#df = df_orig.copy()
```

In [ ]:

```
df.info()
```

Variáveis numéricas

In [ ]:

```
df.dropna(inplace=True)
df = df.drop_duplicates()

df.reset_index(drop = True,inplace=True)
display(df.round().describe())
#display(df.describe())
print('Total de valores NaN: ', df.isna().sum().sum())
print('Dimensão do Dataframe: ', df.shape)
```

In [ ]:

```
# Convertendo variáveis do tipo object para inteiro
## chama função para relacionar as colunas numéricas e categóricas
col = df.columns
col_object=[]
col_numeric=[]
for i in col:
    if df[i].dtypes in [np.object]:
        col_object.append(i)
    elif df[i].dtypes in [np.int64, np.float64]:
        col_numeric.append(i)

print('Variáveis tipo object: ', col_object)

for col in col_object:
    df[col] = pd.Categorical(df[col])
    df[col] = df[col].cat.codes

df.Distrito.describe()
```

Tratamento variáveis Numéricas

Correlação das variáveis - Baixa correlação e Multicolinearidade

In [ ]:

```
df.info()
```

Baixa correlação

In [ ]:

```
#A variável Preco e a correlação com as demais variáveis

df.corr()['Preco'].sort_values().plot(kind = 'bar',yticks = [-.9,-.1,0,.1,.9], mark_rig
ht=False, figsize=(14,5))
```

In [ ]:

```
# Selecionando as variáveis com correlação absoluta maior que |0.1|

s_corr = df.corr()['Preco'].abs()

s_corr = s_corr.loc[s_corr>0.1]
s_corr.sort_values(ascending=False)
l_select_cols = list(s_corr.index)

print('Variáveis consideradas para verificar multicolinearidade:', l_select_cols)
```

In [ ]:

```
# Plotando o mapa de correlação entre as variáveis
sns.set(font_scale = 1)
plt.figure(figsize=(14,10))
plt.title("Matriz de Correlação", fontsize=20)
sns.heatmap(df[l_select_cols].corr(), cbar=True, square= True, fmt='.2f', annot=True, c
map='viridis')
```

In [ ]:

```
# Multicolinearidade - verificando quais as variáveis preditoras estão correlacionadas

df_corr = df[l_select_cols].corr()
df_corr = pd.DataFrame(df_corr)
df_corr.head()
```

In [ ]:

```
# Seleção das variáveis preditoras correlacionadas e criação do dataframe com a respect
iva correlação com Preco
df_corr['indice'] = df_corr.index
df_corr = df_corr.melt(id_vars='indice')
df_corr = df_corr.sort_values(by = 'indice',
                                ascending=False)[(abs(round(df_corr['value'],
                                                            2))>=.90)].loc[df_corr['i
ndice']!= df_corr['variable']]

columns = list(df_corr['indice'].unique())
columns.append('Preco')
df_corr_preco = pd.DataFrame(df[columns].corr()['Preco'].abs()).reset_index().sort_valu
es(by='Preco',ascending=False)
df_corr_preco
```

In [ ]:

```
# Dentre essas variáveis, serão excluídas aquelas de menor grau de correlação em relaçã
o a variável Preco.
# Com a função .merge() serão acrescentadas colunas referentes a variável e sua correla
ção com a preco
df_corr = pd.merge(df_corr, df_corr_preco, how = 'left', left_on='indice', right_on =
'index')
df_corr = pd.merge(df_corr, df_corr_preco, how = 'left', left_on='variable', right_on =
'index')
df_corr.head()
```

In [ ]:

```
# Agora são criadas colunas que irão selecionar quais as que tem maior correlação e as
 de menor correlação
df_corr["manter"] = np.where(df_corr['Preco_x']>df_corr['Preco_y'],df_corr['index_x'],
df_corr['index_y'])
df_corr["excluir"] = np.where(df_corr['Preco_x']<df_corr['Preco_y'],df_corr['index_x'],
df_corr['index_y'])
df_corr.head()
```

In [ ]:

```
# A seguir é aplicado método para listar quais as variáveis que devem ser mantidas e qu
ais devem ser excluídas
# as variáveis a serem excluidas serão subtraídas da lista de variáveis que tinham corr
elação maior que 0.1

l_manter = list(set(df_corr["manter"].unique())-set(df_corr["excluir"].unique()))
l_excluir = list(set(df_corr["excluir"].unique())-set(l_manter))
l_select_cols = list(set(l_select_cols)-set(l_excluir))
print(l_manter)
print('Colunas mantidas:', l_select_cols,'\n\nColunas a serem excluídas por Multicoline
aridade:',l_excluir)
```

In [ ]:

```
# Plotando o mapa de correlação entre as variáveis
l_select_cols = list(df[l_select_cols].corr()['Preco'].abs().sort_values(ascending=Fals
e).index)

sns.set(font_scale = 1.5)
plt.figure(figsize=(14,11))
plt.title("Matriz de Correlação", fontsize=20)
sns.heatmap(df[l_select_cols].corr(), cbar=True, square= True, fmt='.2f', annot=True,
            annot_kws={'size':15}, cmap='viridis')
```

In [ ]:

```
# Selecionando variáveis da lista l_select_cols:

df=df[l_select_cols]
```

In [ ]:

```
l_select_cols
```

In [ ]:

```
#df=df_orig[l_select_cols].copy()
```

Análise variável alvo: Preço

In [ ]:

```
sns.set(font_scale = 2)
plt.figure(figsize=(12,4))
print(sns.boxplot(data=df, x='Preco'))

sns.set(font_scale = 2)
plt.figure(figsize=(12,4))
sns.displot(data=df, x='Preco', kde=True,height=8)

df['Preco'].describe().round(2).astype(str)
```

In [ ]:

```
df_o = df.copy()
```

In [ ]:

```
# Criando novo dataframe com base logatmica para variáveis com distribuição assimétrica

df_ln = df.copy()
cols_ln = simetric(df_ln, l_select_cols)
```

In [ ]:

```
df_ln.info()
```

In [ ]:

```
sns.set(font_scale = 2)
plt.figure(figsize=(12,4))
print(sns.boxplot(data=df_ln, x='Preco'))

sns.set(font_scale = 2)
plt.figure(figsize=(12,4))
sns.displot(data=df_ln, x='Preco', kde=True,height=8)

df_ln['Preco'].describe().round(2).astype(str)
```

In [ ]:

```
# Plotando o mapa de correlação entre as variáveis
df_ln=df_ln[['Preco', 'Area', 'Garagem', 'Condo', 'Banheiros',
        'Mais de 20_SM_%','Quartos', 'Longitude','População (2010)', 'Piscina','Densidade Demográfica (Hab/km²)']]

sns.set(font_scale = 1.5)
plt.figure(figsize=(15,12))
plt.title("Matriz de Correlação", fontsize=20)
sns.heatmap(df_ln.corr(), cbar=True, square= True, fmt='.2f', annot=True, annot_kws={'size':15}, cmap='viridis')
```

Tratamento de Outliers

In [ ]:

```
# A variável Area é a que mais influencia no Preco.
df_out = df.copy()
df_ln_out = df_ln.copy()

df['Preco_Area'] = round(df['Preco'] / df['Area'],2)
df_ln['Preco_Area'] = round(df_ln['Preco'] / df_ln['Area'],2)

sns.displot(data=df, x='Preco_Area', kde=True,height=4),
sns.displot(data=df_ln, x='Preco_Area', kde=True,height=4)
```

In [ ]:

```
plt.figure(figsize=(12,3))
print(sns.boxplot(data=df, x='Preco_Area'))
```

In [ ]:

```
plt.figure(figsize=(12,3))
print(sns.boxplot(data=df_ln, x='Preco_Area'))
```

In [ ]:

```
# Aplicação da função outlier_iqr()
cols = ['Preco_Area']
low_out,upp_out = outlier_iqr(df, cols)

cols = ['Preco_Area']
low_out_ln,upp_out_ln = outlier_iqr(df_ln, cols)
```

In [ ]:

```
# Cálculo do Intervalo Interquatílico (IQR) na variável Preco_Area nos datasets df e df
_ln:

df = df[df['Preco_Area'] > low_out]
df = df[df['Preco_Area'] < upp_out]

df_ln = df_ln[df_ln['Preco_Area'] > low_out_ln]
df_ln = df_ln[df_ln['Preco_Area'] < upp_out_ln]



df.drop(columns = ['Preco_Area'], inplace = True)
df_ln.drop(columns = ['Preco_Area'], inplace = True)

print('Shape do df:', df.shape)
print('Shape do df_ln:', df_ln.shape)
```

In [ ]:

```
# Aplicando RobustScaler em df e df_out
cols = list(df.columns)
df_r = RobustScaler().fit_transform(df)
df_r = pd.DataFrame(df_r, columns=cols)
df= df_r.copy()

cols = list(df_out.columns)
df_r = RobustScaler().fit_transform(df_out)
df_r = pd.DataFrame(df_r, columns=cols)
df_out = df_r.copy()


df_out.head()
```

In [ ]:

```
# Excluindo dados duplicados

df_ = df.drop_duplicates()
df_ln = df_ln.drop_duplicates()
df_out = df_out.drop_duplicates()
df_ln_out = df_ln_out.drop_duplicates()

#Verificando se há dados nulos
print('Totais de dados nulos\ndf: %s \ndf_ln: %s \ndf_out: %s  \ndf_ln_out: %s'
      '\n\nDimensão dos DataFrames\ndf: %s \ndf_ln: %s \ndf_out: %s \ndl_ln_out:%s'
      %(df_.isna().sum().sum(), df_ln.isna().sum().sum(), df_out.isna().sum().sum(), df
_ln_out.isna().sum().sum(),
        df.shape,df_ln.shape, df_out.shape, df_ln_out.shape))
```

In [ ]:

```
df_.info()
```

In [ ]:

```
df_.to_csv('dataset_processado_SP.csv')
df_ln.to_csv('dataset_processado_SP_ln.csv')
df_out.to_csv('dataset_processado_SP_out.csv')
df_ln_out.to_csv('dataset_processado_SP_ln_out.csv')
```

CRIAÇÃO E AVALIAÇÃO DE MODELOS DE MACHINE LEARNING

In [ ]:

```
#df_ = pd.read_csv('dataset_processado_SP.csv', index_col=0)
#df_ln = pd.read_csv('dataset_processado_SP_ln.csv', index_col=0)
#df_out = pd.read_csv('dataset_processado_SP_out.csv', index_col=0)
#df_ln_out = pd.read_csv('dataset_processado_SP_ln_out.csv', index_col=0)
```

In [ ]:

```
def Reg_Linear(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)
    parameters = {'n_jobs':[-1,1,2,3,5,10,25], "fit_intercept": [True, False],'normaliz
e':[False, True]}
    # define modelo/ estimador
    model = LinearRegression()
    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters, scoring='neg_mea
n_squared_error',
                                cv=kf, random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['Regressão_Linear'] = scores
    scores_map[dataset].update(d_scores)

    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
# Função Ridge
def Ridge_(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)
    parameters = {'alpha':[0.0001,0.001,0.01,0.05,0.1,1, 10,100],'max_iter':[10,20,50,1
00,200,300,500,700,1000,2000,5000,10000],
                  'normalize':[False, True], 'fit_intercept':[False, True]}
    # define modelo/ estimador
    model = Ridge()

    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters,scoring='neg_mean
_squared_error',
                                cv=kf, random_state=rs)

    #fit no Rand search
    rand_cv.fit(train_x,train_y)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['Ridge'] = scores
    scores_map[dataset].update(d_scores)
    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def Lasso_(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)
    parameters = {'alpha':[0.0001,0.001,0.01,0.05,0.1,1, 10,100],'max_iter':[10,20,50,1
00,200,300,500,700,1000,2000,5000,10000],
                  'normalize':[False, True], 'fit_intercept':[False, True]}

    # define modelo/ estimador
    model = Lasso()
    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters,scoring='neg_mean
_squared_error',
                                cv=kf, random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores = {}
    scores_map['Lasso'] = scores
    scores_map[dataset].update(d_scores)
    print(scores)
    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)

    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def Elastic_Net(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)
    parameters = {'alpha':[0.0001,0.001,0.01,0.05,0.1,1, 10,100],'max_iter':[10,20,50,1
00,200,300,500,700,1000,2000,5000,10000],
                  'l1_ratio': np.arange(0.0, 1.0, 0.1),'fit_intercept':[False, True],'n
ormalize':[False, True]}
    # define modelo/ estimador
    model = ElasticNet()

    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions= parameters,cv=kf, random_st
ate=rs)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)

    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['ElasticNet'] = scores
    scores_map[dataset].update(d_scores)

    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def Random_Forest_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)

    parameters={'n_estimators':[50,100,150,200], 'max_depth':[1,2,3,5,6,7,8,9,10,15,20,
None],
                'min_samples_leaf':randint(32,128), "min_samples_split":randint(32,128
)}
    # define modelo/ estimador
    model = RandomForestRegressor()
    # definindo Rand search
    rand_cv= RandomizedSearchCV(model, param_distributions=parameters,
                                scoring='neg_mean_squared_error',cv=kf, random_state=rs
)
    #fit no Rand search
    rand_cv.fit(train_x,train_y)

    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['RandomForestReg'] = scores
    scores_map[dataset].update(d_scores)

    # melhor estimador
    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def Decision_Tree_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)
    parameters = {'max_depth':[3,5,10,15,20,30,None],'min_samples_leaf':randint(32, 128
),
                  "min_samples_split":randint(32,128)}

    model = DecisionTreeRegressor()
    rand_cv = RandomizedSearchCV(model, cv=kf, param_distributions=parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)

    #fit no Rand search
    rand_cv.fit(train_x, train_y)
    print("Melhor classificação :", rand_cv.best_estimator_)
    scores = cross_val_score(model,x, y, cv=kf)
    d_scores={}
    d_scores['DecisionTreeReg'] = scores
    scores_map[dataset].update(d_scores)
    # melhor modelo

    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100

    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def K_Neighbors_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)
    parameters = {"n_neighbors" : [2,3,4,5,6,7,8,9,10,12,15],'weights':['uniform', 'dis
tance'],
                  'algorithm':['auto', 'ball_tree', 'kd_tree', 'brute'], 'p':[1,2]}

    model = KNeighborsRegressor()
    rand_cv = RandomizedSearchCV(model, cv=kf, param_distributions=parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x, train_y)

    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['KNeighborsReg'] = scores
    scores_map[dataset].update(d_scores)

    print("Melhor classificação :", rand_cv.best_estimator_)
    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100
    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def Gradient_Boosting_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    dataframe.dropna(inplace=True)
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)

    model = GradientBoostingRegressor()
    parameters={'n_estimators':[50,100, 150,200,250],
                'learning_rate':[0.001,0.01,0.05,0.1,1,5,10],
                "min_samples_split" : randint(32, 128),  "max_depth" : [3, 5, 10, 15, 2
0, 30, None],
                "min_samples_leaf" : randint(32, 128)}

    rand_cv = RandomizedSearchCV(model, cv=kf, param_distributions=parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x, train_y)
    print("Melhor classificação:", rand_cv.best_estimator_)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['GradientBoostingReg'] = scores
    scores_map[dataset].update(d_scores)

    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100

    modelos[dataset].append(modelo)
    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test],
                  'R2 (Acurácia) - Treina':[score_train],'R2 (Acurácia) - Teste': [scor
e_test]}

    return modelos, scores_map,resultados
```

In [ ]:

```
def AdaBoost_Regressor(dataset,dataframe,modelos, scores_map, kf, rs):
    dataframe.dropna(inplace=True)
    x = dataframe.drop(columns='Preco')
    y = dataframe['Preco']
    train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,random_stat
e=rs)

    model = AdaBoostRegressor()
    parameters={'n_estimators':[50,100, 150,200,250],
                'learning_rate':[0.001,0.01,0.05,0.1,1,5,10,100],
                'loss':['linear', 'square', 'exponential']}
    rand_cv = RandomizedSearchCV(model, cv=kf,param_distributions= parameters,
                                 scoring='neg_mean_squared_error', random_state=rs)
    #fit no Rand search
    rand_cv.fit(train_x, train_y)
    print("Melhor classificação:", rand_cv.best_estimator_)
    scores = cross_val_score(model, x, y, cv=kf)
    d_scores={}
    d_scores['AdaBoostReg'] = scores
    scores_map[dataset].update(d_scores)

    # melhor modelo
    modelo = rand_cv.best_estimator_
    modelo.fit(train_x,train_y)

    y_pred = modelo.predict(train_x)
    # Calcula a métrica Root Mean Squared Error
    rmse_train=sqrt(mean_squared_error(train_y,y_pred))
    #Calcula a métrica Mean Absolute Error
    mae_train = mean_absolute_error(train_y,y_pred)
    score_train = modelo.score(train_x, train_y)*100

    modelo.fit(train_x,train_y)
    y_pred = modelo.predict(test_x)
    rmse_test=sqrt(mean_squared_error(test_y,y_pred))
    mae_test = mean_absolute_error(test_y,y_pred)
    score_test = modelo.score(test_x, test_y)*100

    modelos[dataset].append(modelo)

    resultados = {'Dataframe': [dataset], 'Modelo': [modelo], 'MAE - Treina':[mae_train
], 'MAE - Teste':[mae_test],
                  'RMSE - Treina': [rmse_train],'RMSE - Teste': [rmse_test], 'R2 (Acurá
cia) - Treina':[score_train],
                  'R2 (Acurácia) - Teste': [score_test]}

    return modelos, scores_map,resultados
```

Aplicando as funções de ML

In [ ]:

```
modelos={'df':[],'df_ln':[],'df_out':[],'df_ln_out':[]}
scores_map = {'df':{},'df_ln':{},'df_out':{},'df_ln_out':{}}
kf = KFold(n_splits=5, shuffle=True)
rs = 2000

ml_functions = ['K_Neighbors_Regressor', 'Random_Forest_Regressor','Decision_Tree_Regre
ssor', 'Reg_Linear',
                'Ridge_', 'Lasso_','Elastic_Net', 'AdaBoost_Regressor', 'Gradient_Boost
ing_Regressor']

datasets = {'df':df, 'df_ln':df_ln, 'df_out':df_out, 'df_ln_out':df_ln_out}


df_compara = pd.DataFrame(columns = ['Dataframe', 'Modelo', 'MAE - Treina', 'MAE - Test
e', 'RMSE - Treina',
                                     'RMSE - Teste', 'R2 (Acurácia) - Treina', 'R2 (Acur
ácia) - Teste'])

for function in ml_functions:
    for dataset,dataframe in datasets.items():
            modelos, scores_map,resultados = globals()[function](dataset,dataframe,mode
los, scores_map, kf,rs)
            df_mod=pd.DataFrame(data = resultados)
            df_compara=pd.concat([df_compara,df_mod])
```

In [ ]:

```
df_compara.sort_values(by=['R2 (Acurácia) - Teste'], ascending=False, inplace=True)
d_modelo = {df_compara.iloc[0, 0]: df_compara.iloc[0, 5]}
df_compara['Modelos'] = df_compara['Modelo'].astype(str)
df_compara[['Dataframe', 'MAE - Treina', 'MAE - Teste', 'RMSE - Treina',
            'RMSE - Teste', 'R2 (Acurácia) - Treina', 'R2 (Acurácia) - Teste','Modelos'
]].head(30)
```

In [ ]:

```
df_compara.to_csv('resultados02-4.csv')
```

Selecionando os modelos

In [ ]:

```
d_dfs_scores={}
d_dfs_scores['Grad_Boost_df_ln'] = scores_map['df_ln']['GradientBoostingReg']
d_dfs_scores['GradBoost_dfln_out'] = scores_map['df_ln_out']['GradientBoostingReg']
d_dfs_scores['DecTree_df_ln'] = scores_map['df_ln']['DecisionTreeReg']
d_dfs_scores['Grad_Boost_df'] = scores_map['df']['GradientBoostingReg']

d_dfs_scores['RandomForest_df_ln'] = scores_map['df_ln']['RandomForestReg']
d_dfs_scores['KNN_Regr_df_ln'] = scores_map['df_ln']['KNeighborsReg']


plt.figure(figsize=(20, 12))
df_scores_map = pd.DataFrame(d_dfs_scores)
sns.boxplot(data=df_scores_map)
```

Análise de resíduos - Melhor modelo

In [ ]:

```
#Relação de modelos
modelo_sel = [df_compara.iloc[0, 1], df_compara.iloc[5, 1]]

df_ln_pred = df_ln.copy()

modelo=modelo_sel[0]

x = df_ln.drop(columns='Preco')
y = df_ln['Preco']
# Treinando o modelo
modelo.fit(x,y)
y_pred = modelo.predict(x)

df_ln_pred['preco_predito'] = y_pred

# Calcula a métrica Root Mean Squared Error
rmse=round(sqrt(mean_squared_error(y,y_pred)),4)
#Calcula a métrica Mean Absolute Error
mae = round(mean_absolute_error(y,y_pred),4)
score = round(modelo.score(x, y),4)

print("Dataframe: df_ln\nModelo: GradientBoostingRegressor\nMAE:%s \nRMSE: %s\nR²:%s" %
(mae,rmse, score))
plotar_grafico(y,y_pred)
```

Comparação resultados

In [ ]:

```
modelo=modelo_sel[1]

x = df_ln.drop(columns='Preco')
y = df_ln['Preco']

# Treinando o modelo
modelo.fit(x,y)
y_pred = modelo.predict(x)

# Calcula a métrica Root Mean Squared Error
rmse=round(sqrt(mean_squared_error(y,y_pred)),4)
#Calcula a métrica Mean Absolute Error
mae = round(mean_absolute_error(y,y_pred),4)
score = round(modelo.score(x, y),4)

print("Dataframe: df_ln\nModelo: RandomForestRegressor\nMAE:%s \nRMSE: %s\nR²:%s" %(mae
,rmse, score))

plotar_grafico(y,y_pred)
```

Apresentação de Resultado

Precisão por distritos e preços médios

In [ ]:

```
df_ln_pred.rename(columns={'Preco':'Preco_ln'}, inplace=True)

df_results = df_orig[df_orig.index.isin(df_ln_pred.index)]

df_results = pd.concat([df_results,df_ln_pred[['Preco_ln','preco_predito']]], axis=1)

df_ln_pred.rename(columns={'Preco_ln':'Preco'}, inplace=True)
df_ln_pred.drop(columns=['preco_predito'], inplace=True)

df_results['preco_predito_x'] = np.exp(df_results['preco_predito'])
df_results['Precisao'] = np.where(df_results['preco_predito_x']/df_results['Preco']>1,
                                  df_results['Preco']/df_results['preco_predito_x']*100,
                                   df_results['preco_predito_x']/df_results['Preco']*100
)

df_results.shape
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df_results.groupby('Distrito')['Precisao'].mean().reset_index()

df_group = grouped.copy()

grouped = grouped.sort_values(by = 'Precisao', ascending = True).head()

plt.figure(figsize = (5,4))
sns.set(font_scale = 1.4)
ax = sns.histplot(x='Precisao', y='Distrito', data = grouped,bins=100)
print("Precisão: Distritos com as piores previsões")
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df_results.groupby('Distrito')['Precisao'].mean().reset_index()

#df_group = grouped.copy()

grouped = grouped.sort_values(by = 'Precisao', ascending = False).head(5)

plt.figure(figsize = (5,4))
sns.set(font_scale = 1.40)
ax = sns.histplot(x='Precisao', y='Distrito', data = grouped, bins=100)
print("Precisão: Distritos com as melhores previsões")
```

In [ ]:

```
# Distribuição dos imóveis na cidade de São Paulo

grouped = df_results.groupby('Distrito')['Preco'].mean().reset_index()

df_group = pd.merge(df_group, grouped, how='inner', on='Distrito')

df_group.rename(columns={'Preco':'Preco_Medio'}, inplace=True)
df_group = pd.merge(df_group, df_medias, how='inner', on='Distrito')

df_group.to_csv('ap_resultado.csv')
```

In [ ]:

```
# Criando Regras de zona

df_group['Zona']= np.where(df_group.media_Longitude>df_group.media_Longitude.quantile(
0.70),'Centro/Leste',
                                    np.where(df_group.media_Latitude<df_group.media_Lat
itude.median(),'Sul',
                                                        'Centro/Norte'))
df_group['Zona']= np.where(df_group.media_Longitude>df_group.media_Longitude.quantile(
0.80),'Leste',df_group['Zona'])
```

In [ ]:

```
df_group.describe()
```

Mapa de resultados

In [ ]:

```
#Preparando Mapa de resultados

sul_group = folium.FeatureGroup(name='Sul')
leste_group = folium.FeatureGroup(name='Leste')
norte_group = folium.FeatureGroup(name='Centro/Norte')
centro_leste_group = folium.FeatureGroup(name='Centro/Leste')
m = folium.Map(location=[-23.55,-46.63], zoom_start=10)
ms = MarkerCluster()
ml = MarkerCluster()
mn = MarkerCluster()
mcl = MarkerCluster()
for dis,acur,preco,lat,long, zone in df_group.values:
    # coordinates to locate your marker
    COORDINATE = [lat,long]
    if zone == "Sul":
        ms.add_child(folium.Marker(location=COORDINATE,
                        popup=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preço Mé
dio aptos R$ '+str(round(preco,2)),
                        tooltip=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preço
 Médio aptos: R$ '+str(round(preco,2)),
                        icon=folium.Icon(icon='book'))).add_to(sul_group)
    elif zone == "Leste":
        ml.add_child(folium.Marker(location=COORDINATE,
                        popup=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preço Mé
dio aptos R$ '+str(round(preco,2)),
                        tooltip=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preço
 Médio aptos: R$ '+str(round(preco,2)),
                        icon=folium.Icon(icon='book'))).add_to(leste_group)
    elif zone == "Centro/Norte":
        mn.add_child(folium.Marker(location=COORDINATE,
                        popup=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preco Mé
dio aptos R$ '+str(round(preco,2)),
                        tooltip=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preco
 Médio aptos: R$ '+str(round(preco,2)),
                        icon=folium.Icon(icon='book'))).add_to(norte_group)
    else:
        mcl.add_child(folium.Marker(location=COORDINATE,
                        popup=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preço Mé
dio aptos R$ '+str(round(preco,2)),
                        tooltip=str(dis)+', Precisão '+str(round(acur,2))+ '%' +'. Preço
 Médio aptos: R$ '+str(round(preco,2)),
                        icon=folium.Icon(icon='book'))).add_to(centro_leste_group)

m.add_child(sul_group)
m.add_child(leste_group)
m.add_child(norte_group)
m.add_child(centro_leste_group)
# turn on layer control
m.add_child(folium.map.LayerControl(collapsed=False))
m.save("mapa_resultados.html")
m
```

In [ ]:

```
# Dicionário modelos. Chaves (nomes dos dataframes) e valor (modelo de ML com os respec
tivos melhores parâmetros calculados)
modelos
```

FIM